NASA CR-134476

NASIS DATA BASE MANAGEMENT SYSTEM - IBM 360/370 OS MVT IMPLEMENTATION

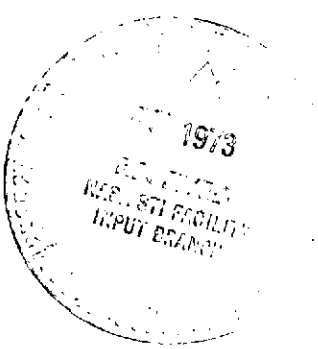
VII - DATA BASE ADMINISTRATOR USER'S GUIDE

NEOTERICS, INC.

prepared for

NATIONAL AERONAUTICS AND SPACE ADMINISTRATION

NASA Lewis Research Center
Contract NAS 3-14979



1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.
NASA CR-134476		

4. Title and Subtitle	5. Report Date
NASIS DATA BASE MANAGEMENT SYSTEM - IBM 360/370 OS MVT IMPLEMENTATION VII - DATA BASE ADMINISTRATOR USER'S GUIDE	September 1973
	6. Performing Organization Code
7. Author(s)	8. Performing Organization Report No. None
	10. Work Unit No.
9. Performing Organization Name and Address Neoterics, Inc. 2800 Euclid Avenue Cleveland, Ohio 44115	11. Contract or Grant No. NAS 3-14979
	13. Type of Report and Period Covered Contractor Report
12. Sponsoring Agency Name and Address National Aeronautics and Space Administration Washington, D.C. 20546	14. Sponsoring Agency Code

15. Supplementary Notes
Final Report. Project Manager, Charles M. Goldstein, Computer Services Division, NASA Lewis Research Center, Cleveland, Ohio

16. Abstract

The NASIS development workbook contains all the required system documentation. The workbook includes the following seven volumes:

I - Installation Standards (CR-134470)
II - Overviews (CR-134471)
III - Data Set Specifications (CR-134472)
IV - Program Design Specifications (CR-134473)
V - Retrieval Command System Reference Manual (CR-134474)
VI - NASIS Message File (CR-134475)
VII - Data Base Administrator User's Guide (CR-134476)

17. Key Words (Suggested by Author(s))	18. Distribution Statement
	Unclassified - unlimited

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of Pages	22. Price*
Unclassified	Unclassified	179	

* For sale by the National Technical Information Service, Springfield, Virginia 22151

TABLE OF CONTENTS

TOPIC E- TERMINAL SUPPORT

TOPIC G - USAGE STATISTICS

TOPIC A.2 - MT/T OPERATOR'S GUIDE

I. INTRODUCTION

The single program that controls NASIS when the MT/T version of that system is running is called the MT/T Monitor. The monitor is the only part of NASIS with which the MT/T Operator communicates.

To communicate with the monitor simply depress the ATTENTION key. The monitor will prompt you with a time-stamped question mark, for example:

10:25 ?

and unlock the keyboard. Note that while your keyboard is unlocked, NASIS is stopped. Waste no time in entering commands and never, never leave your terminal sitting with its keyboard unlocked.

II. MONITOR COMMANDS

The monitor commands are comprised of a command name and, in some cases, additional operands. The monitor, when reading commands, recognizes three "special" characters--two delimiters: (separators between command names and/or operands) comma and blank, and a character which may enclose an operand to denote that that operand has "special" characters within it: the quote mark. The delimiters behave slightly differently--a string of contiguous blanks is interpeted as one delimiter, but two contiguous commas are interpeted as two delimiters, and so forth. If you have to put blanks, commas or quotes within an operand, you must surround that operand with quote marks. In addition, if there are enclosed quotes, they must be paired inside the operand. For example

'don''t let this confuse you, it''s not really that difficult'

is a valid quoted string containing embedded commas, blanks and quote marks.

MSG NASISID,TEXT Sufficient Abbreviation (S.A.): M

This command sends the message specified by the TEXT operand to the user who is on NASIS under the userid specified by the operand NASISID. Remember to surround the message text with quote marks if it contains commas, quote marks, or imbedded spaces. Example:

M NEO1,'HERE''S A MESSAGE.'

BCST TEXT S.A.: B

This command sends the message specified by the TEXT operand to all the users logged on to NASIS. Example:

```
BC 'DATACELL IS DOWN NSIC notAVAILABLE.'
```

FORCE NASISID S.A.: F

This command is used to terminate a NASIS user. The user (identified by NASISID) is sent the message "*** TASK DELETED BY FORCE ***" and then logged off. Example:

```
F NEO1
```

KILL NASISID S.A.: K

This command is used when FORCE fails. The KILL command may be reentered several times. The user (if the KILL works) will receive a program interrupt five at location zero, so you may ignore the message about that event. Example:

```
KI NEO1
```

SHUTDOWN TIME S.A.: S

This command terminates NASIS. The TIME operand specifies how long to wait before actually terminating the system (default is five minutes). If the time specified is zero minutes NASIS is terminated immediately. This zero-time shutdown should be used only when absolutely necessary because it doesn't give warning to the users. Normally, both you and the users get a message stating the time-of-day when the system will shut itself down. Should you change your mind about the shutdown enter another shutdown to override the previous one. (Only the last SHUTDOWN command entered has any effect.) Example:

```
S 30            (To terminate NASIS in a half-hour)
```

LIMIT TERM,# S.A.: I

This command allows you to limit the number of users of various sorts allowed on NASIS and to limit some of the resources of NASIS itself. The TERM operand is either a "class" of NASISIDs (defined as the first two characters of the NASISID) or one of the keywords "USERS", "PRINTS", "SEARCHES", "SORTS" or "RECORDS". The keyword "USERS" is used to limit the total number of users allowed on NASIS and is the default value assumed if TERM is omitted. Keyword "SEARCHES" limits the size of a set a NASIS user may search on, "PRINTS" limits the size of a set he may print and so on. If the TERM operand consists of exactly two characters it is assumed to be a class name and the number of NASISIDs of that class allowed on NASIS will be limited. If the TERM operand consists of any other number of characters than two, it is assumed to be a keyword or a part of a keyword. If the # operand is defaulted, the value 32767 is used. If the # operand is entered, TERM must be also entered, even if you use just a comma to default it. Examples:

```
LIMIT ,20       (Limit total number of users to 20)
L S,50          (Limit search set size to 50)
```

```
LI NE,2        (Limit "NE" NASISIDS to 2)
```

```
USERS          S.A.: U
```

This command lists all the NASISIDs of the users currently using NASIS. Only those users completely logged on are listed, if there are users in the process of getting on, they will not show up on the list from a USERS.

```
NUSERS         S.A.: N c/:N/: N/
```

This command tells you how many users are currently using NASIS. Unlike USERS, this command also tallies the users who are in the process of logging on.

```
NEWS "OFF"|TEXT          S.A.: NE
```

This command is used to control the sending and composition of the "news" which is sent to each user as he logs on to NASIS. Entering "OFF" as the operand terminates the sending of all news and deletes all the text from the news buffer. Entering anything but "OFF" causes whatever you enter to be added as the last line to whatever is already in the news buffer. If you enter no operands at all to NEWS, it will add a carriage-return to the end of the news buffer. Examples:

```
NEWS OFF          (Kills the sending of news)
NEWS 'THIS LINE WILL GO AT THE END OF THE BUFFER.'
```

```
STATS "ON/"OFF"              S.A. ST
```

When this command with operand OFF is encountered, the Monitor turns on an indicator telling NASIS not to take usage statistics. If ON is entered as the operand, that indicator is turned off. NOTE: This command may only be entered via the "NASIS.COMMANDS(0)" dataset.

APPENDIX A. - COMMAND SUMMARY

COMMAND	OPERANDS	FUNCTION
MSG	NASISID,TEXT	Send message to specified user.
BCST	TEXT	Send message to all users.
FORCE	NASISID	Get rid of a user.
KILL	NASISID	Really get rid of a user.
SHUTDOWN	TIME	Terminate NASIS.
LIMIT	TERM,#	Limit NASIS users or resources.
USERS		List current NASIS users.
NUSERS		Count current NASIS users.
NEWS	"OFF"|TEXT	Turn off or add to news text.
STATS	"ON"/"OFF"	Set usage statistics mode.

TOPIC B.1 - CONVERSION, VALIDATION, AND FORMATTING ROUTINES

I. INTRODUCTION

The design of the NASIS system provides for three types of user-written routines to perform special processing unique to a particular field. A "user" is a mainline programmer for the specific data base; such as the data base administrator. These routines are classified as conversion, validation and formatting.

The DBPL/I statements used in the NASIS system provide for updating and retrieving from a data base. The data is always assumed to be character strings. The ability to specify Conversion, Validation and formatting routines is provided, allow for massaging field data and still meet the DBPL/I character string requirement.

The Conversion routine is used to alter character string input to any desired form. The Validation routine is used either to verify the results of a Conversion routine or to verify the character string input.

The Formatting routine is used to alter the internal stored data back to a character string.

A. CONVERSION Routine

The CONVERSION routine is called by the data base executive, DBPAC, to convert the data passed by the user in a DBPL/I statement from an EBCDIC character string to some other type of representation for storage on a file. The CONVERSION routine is invoked by all DBPL/I statements that place data, by field name, onto the data base.

B. VALIDATION Routine

The VALIDATION routine is call of theed immediately after the call CONVERSION routine. The function of this routine is to verify data input for storage on the data base, via the rules specified by the user of this field. A VALIDATION routine may be present regardless of the presence of a CONVERSION routine. To assist in this evaluation, the NASIS system provides for a validation argument.

C. FORMATTING Routine

The FORMATTING routine is called to change the data read from the data base into the desirable output form. The FORMATTING routine is invoked by all DBPL/I statements that retrieve data, by field name, from the data base. The formatting routine specified for a field will be called whenever the data in that field is retrieved.

A collection of "standard" conversion and formatting routines is provided in the DBEXITS module (Section IV, Topic B.4).

II. CALLING SEQUENCE

In general, these routines are called dynamically, by name. They must have been link-edited with the current Retrieval system and be capable of accepting a PL/I formatted parameter list.

A. CONVERSION Routine

The format of the CALL statement used by DBPAC to invoke the CONVERSION routine is as follows:

```
CALL    rtnname  (input-data, output-area,
error-bit);
```

where:

"rtnname" identifies the particular routine to be called, as specified in the field descriptor. It is the routine's procedure name or an entry point.

"input-data" is a varying length character string, maximum length equal to 4000, into which DBPAC has placed the input data value.

"output-area" is a varying length character string, maximum length equal to 4000, initialized to null, into which the exit routine places the converted data value.

"error-bit" is a bit switch, initialized to one (1), which is set to zero (0) if there were no errors uncovered in the conversion, or one (1) if errors were detected. The burden of setting the switch to zero (0) is with the

CONVERSION routine.

B. VALIDATION Routine

The format of the CALL statement used by DBPAC to invoke the VALIDATION routine is as follows:

CALL rtnname (input-data, output-area, error-bit, argument);

where:

"rtnname" identifies the particular routine to be called, as specified in the field descriptor. It is the routine's procedure name or an entry point.

"input-data" is a varying length character string, maximum length equal to 4000, into which DBPAC places the input data value after conversion.

"output-area" is a varying length character string, maximum length equal to 4000, initialized to null, into which the exit routine places the validated data value.

"error-bit" is a bit switch, initialized to one (1), which is set to zero (0) if there were no errors encountered in the validation, or one (1) if errors were detected. The VALIDATION Routine is responsible for setting this switch.

"argument" is a varying-length character string, maximum length equal to 50, into which DBPAC places the validation argument, as read from the appropriate field of the descriptor for this data field.

C. FORMATTING Routine

The format of the CALL statement used by DBPAC to invoke the FORMATTING routine is as follows:

CALL rtnname (input-data, output-area);

where:

"rtnname" identifies the particular routine to be called, as specified in the field

descriptors. It is the routine's procedure name or an entry point.

"input-data" is a varying length character string, maximum length equal to 4000, into which DEPAC places the data value read from the data base.

"output-area" is a varying length character string, maximum length equal to 4000, initialized to null, into which the exit routine places the formatted data value.

III. RESTRICTIONS

The routines must heed the following restrictions:

A. The routine can not make any calls to DBPAC (i.e., it should not contain any DBPL/I statements).

B. The routine is the lowest level module; i.e., it does not call any other routines.

C. The routine is written in PL/I and compiled with the same compiler as the Retrieval PL/I modules.

APPENDIX A.

Diagnostic Messages and Codes Produced By the Conversion, Validation, and Formatting Routines.

A. Diagnostic Messages

CALL ERROR: MODULE ******** CANNOT BE LOADED.

This error message is generated if the module named cannot be loaded when called by DBPAC. Ignoring the situation and allowing the system to run may cause unpredictable results.

The most probable reasons for this error are:

1. failure on the part of the user to have the job library containing this program properly DDEFed.

2. inconsistency between the name of the routine as specified in the descriptor file and the name actually used when writing the program.

B. DBPAC Error Codes Associated With the Conversion, Validation, and Formatting Routines

031 KEY FIELD FAILED CONVERSION.

The data value passed to the CONVERSION routine, for the key field of the data base, was found to be in the wrong format.

032 KEY FIELD FAILED VALIDATION.

The data value passed to the VALIDATION routine, for the key field of the data base, was found to be invalid.

053 DATA FIELD FAILED CONVERSION.

The data value passed to the CONVERSION routine, for a data field, was found to be in the wrong format.

054 DATA FIELD FAILED VALIDATION.

The data value passed to the VALIDATION routine, for a data field, was found to be invalid.

APPENDIX B.

Sample Validation Routine

A sample VALIDATION routine is shown below. The function of the routine is to compare the input data value to each of the four byte entries carried in the validation arguments. If a match is found, the routine substitutes a numeric code for the input data value, resets the error bit to accept the field and returns to DBPAC. If no match is found, the routine returns to DBPAC with the error bit set to reject the field.

```
/* THIS IS A VALIDATION ROUTINE FOR THE OPERATION CODES:  */
/*  THE PARAMETERS PASSED ARE:                            */
/*       A= THE INPUT STRING WHICH IS TO BE VALIDATED.    */
/*       B= THE VALUE TO BE RETURNED AFTER VALIDATION.    */
/*       C= THE BIT SWITCH. '0' MEANS PASSED VALIDATION.  */
/*                          '1' MEANS FAILED VALIDATION.  */
/*       D= THE VALIDATION ARGUMENTS.                     */
/*       D IS COMPOSED OF THE FOLLOWING CHARACTER STRING: */
/*         'ADDEADDRCNGEFLDCDELRDELF'                     */
 CHECKOP: PROCEDURE (A,B,C,D);
      DECLARE (A,B,D) CHARACTER(*) VARYING, /*PARAMETERS. */
              C BIT(1);                     /*PARAMETERS. */
      ON ERROR GO TO OUT_DIRTY;
      DO I = 1 TO 21 BY 4;
      IF A = SUBSTR(D,I,4)
      THEN GO TO OUT_CLEAN;
      END;
 OUT_DIRTY: /* IF IT DOES NOT MATCH KEYWORDS IN ARGUMENT. */
      C = '1'B;
      RETURN;
 OUT_CLEAN: /* THE VALIDATION OF OP_CODE WAS SUCCESSFUL.  */
      C = '0'B;
      B = A;
      RETURN;
      END CHECKOP;
```

TOPIC B.2 - DBPL/I LANGUAGE EXTENSION USER'S GUIDE

I. INTRODUCTION

This manual is for PL/I Programmers writing a mainline program that accesses a NASIS data base. The data base organization being used is fully specified in the "NASIS Overview".

All data base access is done by a combination of:

1. an extension of the PL/I language, called DBPL/I, for data base access,
2. a compilation-time source program processor, DB, and
3. execution-time routines DBPAC and DBLIST.

This manual is the specification of the DBPL/I language extension and is the reference manual to the DB preprocessor. Detailed specification of the internals of the DB preprocessor are given in Section IV, Topic B.1 of the DWB, and the details on the execution-time routines are given in the DBPAC Design Specifications (Section IV, Topic B.2 of the DWB). Neither of these two sections are needed for writing, compiling and executing mainline programs; they may be needed for debugging.

Chapter II of this manual discusses the usage of the DB preprocessor. Chapters III through VI are composed of discussions and examples of the different features of DBPL/I and their interrelationships. Chapter VII, "Rules and Syntactic Descriptions", provides a detailed reference to specific information in alphabetical order. Appendix A is a quick reference to DBPL/I syntax.

II. THE PREPROCESSOR

A. Overview

DBPL/I language statements have to be processed at compilation-time. The processing consists of syntax analysis and the generation of PL/I statements CALLing DBPAC to accomplish what the DBPL/I statements signify. This processing is done by the preprocessor stage of the PL/I compiler under control of a preprocessor procedure named DB. A programmer using DBPL/I does not have to write the DB preprocessor or be concerned with the PL/I statements that are generated by it; but he is required to write certain statements in his

source program so that the DB preprocessor is properly invoked by the PL/I compiler for his program. He must also refrain from using certain identifiers which are reserved words for the DB preprocessor's exclusive use.

B. Usage

The statements required to properly invoke the DB preprocessor are illustrated in an example program in Figure 1.

```
1.  FIG_1: PROCEDURE OPTIONS (REENTRANT);
2.    % INCLUDE LISRMAC(DB);
3.    DECLARE REPORT# CHARACTER (13) VARYING;
4.
5.    DB (( ON ERRORFILE(STAR) GO TO NOTE; ))
6.
7.    DB ((
8.      READ FILE(STAR) KEY('67N26508');
9.  GET FILE(STAR) FIELD('REPTNO') INTO(REPORT#);
10.     ))
11.   PUT DATA (REPORT#);
12.   RETURN;
13.
14. NOTE: PUT DATA (STAR.ONCODE);
15. DONE: DB (( FINISH;  ))
16.   END FIG_1;

% INCLUDE(DB);
```

One %INCLUDE DB statement must be written immediately following the external PROCEDURE statement of the compilation. Any PROCEDURE statement attributes could have been used in line 1. The % INCLUDE DB statement must precede all other statements such as line 3.

```
DB((ON ERRORFILE(STAR) GO TO NOTE;))
```

Any DBPL/I statement, such as this ON statement, must be written as a subargument in a DB preprocessor function reference. As many DB statements may be used as required. Any PL/I statements required may be used at lines 3, 4, 6, and 11-14. Lines 7-10 illustrate that more than one DBPL/I statement may be written in one DB statement. However, no non-DBPL/I statements would be permitted within a DB function reference.

```
DB((FINISH;))
```

One DBPL/I FINISH statement must be written following all other D/B statements in the compilation. It will usually be written just preceding the END statement of the external procedure because it generates a RETURN statement. If the statement in line 14 is executed, then the procedure will be terminated by control passing sequentially to the RETURN statement generated for line 15. The label in line 15 is not required, but it would be valid as shown (e.g., line 12 could be: GO TO DONE;).

The DB preprocessor function generates diagnostic comments (see Section III, Topic B.1 of the DWB). When reviewing a compilation, the programmer should first find the summary diagnostic message (DB067) to know how many error diagnostics for which to search.

C. Reserved Words

The FINISH ON-condition is reserved for use by the DB preprocessor. The following identifiers are reserved for the uses specified in this manual or for the DB preprocessor's use:

```
CCLIST
CPLIST
DB
DBEFCBP and all other identifiers beginning
        'DB'
DUPLIST
ERRORFILE
#FIELD
DBPL/I file-names
FINISH
LIST
#LIST
LISTERR
ULIST
UPLIST
#XREF
```

The PL/I HIGH and NULL built-in function names may be used as such in the program, but the names must not be otherwise declared.

III. DATA BASE AND FILES

A. Overview

The DBPL/I language provides statements that enable data to be transmitted between internal

main storage and external storage devices organized as one or more data bases.

B. Data Sets

Each "data set" is a named, labelled collection of related data, subdivided into keyed data set records.

The one "descriptor data set" for a data base stores data describing the information data set(s) and their interrelationships. It is a collection of one or more descriptor regions.

Each "descriptor region" is a collection of descriptor records for an information data set. The first record in a descriptor region is a data set descriptor record. Subsequent records in a descriptor region are field descriptor records.

C. Files

DBPL/I requires a file name to be used for a file. What data set(s) a file name represents is deduced from the file title. Characteristics of a file may be described with keywords, called file attributes, specified for the file name, deduced contextually, or assumed by default.

A "file name" is an identifier specified in the FILE clause of DBPL/I statements. A file name may not exceed the seven-character length limitation for external names. The user must execute a PL/I ALLOCATE statement for the MFCB before executing any DBPL/I statements. For example, to use a DBPL/I file-name "plex" the following statement must be executed:

```
ALLOCATE PLEX;
```

Of course the allocation must be done in a program in which PLEX will be automatically declared because of its use in a DBPL/I statement. If the module where the ALLOCATE is to be done does not otherwise need DBPL/I statements, the following are recommended as a minimum:

```
% INCLUDE LISRMAC(DB);
ALLOCATE PLEX;
DB((CN ERRORFILE(PLEX) SYSTEM;))
DB((FINISH;))
```

A "file title" can be specified for a DBPL/I file

either through the file name or through the character string value of the expression in the TITLE option of a DBPL/I OPEN statement. If a file is OPENed implicitly, or if no TITLE option is specified in the OPEN statement that causes explicit opening of the file, the file title is assumed to be the same as the file name.

A file title, not beginning with a pound sign (#), consists of a six-character left-aligned dataplex identification and a one-character suffix. Which data set(s) the file name represents will be deduced from the file title suffix value as follows:

blank: the identified data base or anchor data set (for physical record operations: GET RECORD or WRITE).

numeric: the particular asscciated data set.

Z-Q: the particular subfile data set.

A-P: the particular index data set.

A pound sign (#), prefixed to a file title, specifies that a file name represents the descriptor region rather than the information data set itself. (This combination may be specified only in the TITLE option of a DBPL/I OPEN statement because it results in an eight-character title.) If the eighth character of a descriptor region title is blank, the file represents only the anchor descriptor region. This facility allows mainline prcgrams tc create, maintain or retrieve from descriptor regions for their own purposes.

File "attributes" for a file name may be specified explicitly in a DBPL/I OPEN statement or assumed by default. Different attributes may be applied in different openings of the same file in a program; at any particular time, the attributes applied by the most recent opening apply to the file name.

D. File Level Statements

DBPL/I provides the OPEN, CLOSE and ON ERROFFILE statements for file level operations. All are optional; a simple mainline may not need any of them. There is nc statement for declaring a

DBPL/I file; the DB preprocessor generates the necessary Mainline File Control Block (MFCB) automatically.

The OPEN and CLOSE statements may be used for any of the purposes indicated in their descriptions in Chapter VII of this manual.

The ON ERRORFILE statement is used to establish a user's error routine in the mainline to which the DBPAC execution routines will return when an error condition (e.g., key not found) occurs on a file. Several ON statements for a file may be executed in a program either before or after the file is opened.

An "error routine" must begin with a statement label (the same label identifier specified in an ON statement). PL/I (or DBPL/I) statements may be written following the label to handle the error. These statements may reference certain fields in the MFCB for assistance in determining the error identity and resuming normal execution. MFCB fields are referenced using a qualified name consisting of the file name and an MFCB field name. The MFCB fields that may be referenced in a file exception routine are as follows:

file-name.ONCODE is a binary integer whose value specifies the exceptional condition. The meanings of the various ONCODE values are in Section III, Topic B.3 of the DWB.

file-name.ONFILE is the current file title.

file-name.ONFIELD is the current field name (when applicable).

file-name.ONRETURN is a label variable set by DBPAC.

An error routine may be terminated in any manner; for certain of the less serious ONCODEs, a GO TO file-name.ONRETURN; statement may be used which transfers control to the statement following the one that raised the exceptional condition.

For a more generalized exception routine for one or more files, the relevant MFCB fields may be referenced using a qualified name consisting of the reserved keyword ERRORFILE and an MFCB field name; e.g., ERRORFILE.ONCODE.

IV. RECORDS

A. Overview

The data items in a data set are arranged in data set records. In this manual, a "physical record" means a single data set record having an internal self-defining, variable-length format, a fixed-length internal key, and the other data items.

The simple term, "record", in this manual means either a logical record or a physical record, depending on content.

The "current record of a file" is the single record having the key value established by the most recent record level operation on the data base component file. It is accessible only by DBPL/I statements; the mainline has no means of addressing it. In a spanned index, the "current record" is actually a "region" of one or more physical records made to behave like one logical record.

B. Record Level Statements

DBPL/I provides the LOCATE, READ, and UNLOCK statements for record level operations. The record level statements cause a record (possibly more than one physical record) to be transmitted between the data set(s) and the current record of a file. The transmission may be immediate (READ or UNLOCK after update) and/or subsequent (LOCATE or READ for update). LOCATE and READ cause automatic file opening, if necessary.

The LOCATE statement is used to create a new current record having a new key for subsequent transmission to the file (no WRITE statement is needed). The LOCATE SUBFILE statement is used to create a new current subrecord.

The READ statement is used to retrieve a record from a file and establish it as the current record of the file. If the record is updated, it is subsequently retransmitted to the file (no REWRITE statement is needed). The READ SUBFILE statement is used to retrieve a subrecord and READ INDEX to retrieve an index record.

The UNLOCK statement releases a locked current record so that other tasks can read it. If the

record was updated, it is retransmitted to the file. The UNLOCK SUBFILE statement releases a locked current subrecord.

C. Physical Record Statements

DBPL/I provides the GET RECORD and WRITE statements for physical records. These are special purpose statements intended for use in a utility mainline for backing up, restoring or reorganizing one particular data set at a time. They may be used only by the owner of the data base.

The GET RECORD moves the current physical record without change to the user's receiving field (for backup purposes).

The WRITE statement transmits a physical record from the mainline without change to a data set (for restoring or reorganizing purposes). WRITE causes automatic file opening, if necessary.

V. FIELDS

A. Overview

The data items in a record are arranged in fields and, optionally, field elements.

A "field" is a data item having a field name, an internal field descriptor and one or more values per record. Since some fields may have multiple values per record, an individual data item is called a field element. This section of the manual relates primarily to anchor, associated and subrecord fields, although the GET INDEX KEY statement may be used for index key fields. Facilities for subfile control fields and for the list-of-keys field in index records are discussed in Chapter VI of this manual.

A "field name" is an eight-character string value identifying a field. A mainline written in terms of a known particular data base may use a character-string constant in string quotes. A more generalized mainline may use an eight-character string variable and assign a value to it from input data or from a descriptor record before using it as a field name. The names of the fields in field descriptor records are given in the Descriptor File Specification.

An "internal field descriptor" is either a field descriptor record in a descriptor region (for data base fields) or an internal descriptor (for descriptor fields). The descriptor record for an anchor field may limit GET access of a FIELD to those users the file owner has authorized. (PUT and REPUT may be used only by the file owner).

A "field element value" is always a varying length character string value in the mainline. (Internally, it may be fixed- or variable-length and character or coded form.) There may be some transformation between the internal value and the mainline value. If the field descriptor names an input validation and/or conversion routine or an output formatting routine, the relevant routines will be invoked automatically when the field is accessed.

The internal value of a field element is null until a value is PUT into it. A GET FIELD statement retrieves a value even if it is null; a null value yields a null mainline string value (unless a formatting routine translates a null internal value to something non-null such as 'NO DATA YET'). To handle such a case, the most general way to retrieve field values is as follows:

```
     DO  I=1  TO MAX(#FIELD(mfcb,fldname),1);
     DB((GET FILE(mfcb)FIELD(fldname)INTO(var);))
     IF LENGTH(var)=0
       THEN GO TO FIELD_EXHAUSTED;
     /*Use field element value in var.*/
     END;
FIELD_EXHAUSTED:
```

Do not attempt GET FIELD more than #FIELD times or something like 'NO DATA YET' will be retrieved after values actually present. The mainline may determine if the field element is null by testing if the length of the mainline string is zero. A REPUT statement replaces an element with a new value which may be a null value.

B. Field Level Statements and Functions

DBPL/I provides the PUT FIELD, GET FIELD and REPUT statements for the creation, retrieval, and maintenance of field elements on anchor and subfile records. #FIELD is a PL/I function provided for obtaining the numbers of elements in a field. The field level statements cause one or

more field elements to be transmitted individually between the current record of a file and a mainline program. A record level statement must have been executed to establish a current record of the file before a field level statement may be executed.

The PUT FIELD statement is used to create a new field element in the current record of the file. It is subsequently transmitted to the file automatically (no WRITE or REWRITE statement is needed).

The GET FIELD statement is used to retrieve a field element from the current record of the file.

The REPUT statement is used to replace an existing primary field element in the current record of the file. It is subsequently retransmitted to the file automatically (no REWRITE statement is needed).

The #FIELD function calculates the number of elements in a field. It may be used to govern GETting of elements or merely to determine if a field has any elements or not.

For a field that may not have multiple elements, the field level statements transmit the single field element value.

The following discussion applies to fields that may have multiple elements. PUTting an element appends it to the right end of the field. GETting of a FIELD element proceeds from left to right and, when the end of the field is passed, null values are generated. REPUTting an element replaces the "current element of the field" which is the element most recently obtained by a GET FIELD from the current record of the file. There is no facility for referring to an element by its position (subscript) in the field. If it is necessary to (re)GET an element that is to the left of the current element, the record may be (re)READ, resetting all of the internal current element counters to the first element of the fields. If it is necessary to maintain field elements in some order depending on their mainline values (rather than the order in which they are entered), the following technique may be used (for ascending sequence):

```
   DECLARE (OLD,NEW) CHARACTER (maxlen) VARYING;
   NEW = expression;
   DB (( READ FILE (name) (positioning);
NEXT_ELEMENT:
   GET FILE (name) FIELD (fieldname) INTO (OLD);
   ));
   IF LENGTH (OLD)    /* IF OLD IS NON-NULL */
     THEN DO;
       IF OLD > NEW          /* GREATER THAN*/
         THEN DO;  /*INSERT ELEMENT */
           DB ((REPUT FILE(name)
             FIELD (fieldname) FROM (NEW); ));
           NEW = OLD;  /* FOR PROPAGATION  */
           END;
       GO TO NEXT_ELEMENT;
       END;
   DB (( PUT FILE (name) FIELD (fieldname) FROM
   (NEW); ));
```

C. Index Field Retrieval

DBPL/I provides a special GET INDEX KEY statement and the #XREF function for retrieval from index records. (Such records may not be explicitly created or maintained by mainline programs). A READ INDEX statement must have been executed to establish a current record of an index before a GET INDEX KEY or #XREF may be executed.

The GET INDEX KEY statement is used to retrieve the index key field value from the current record of the index.

The #XREF function calculates the number of cross references (anchor or subfile key elements) in the current record (region) of the index.

The GET FIELD statement and #FIELD will not work on index record fields. An index record RECLEN field cannot be retrieved (it doesn't mean much in a spanned index). The GET INDEX KEY statement is provided for the index key field. #XREF is provided (instead of #FIELD) for the cross reference field element count. The GET INDEX LIST SET statement (see section VI.B below) retrieves the whole cross reference list (instead of an element at a time).

VI. LISTS

A. Overview

A "list" of keys is a collection of ascending internal key elements in main storage, identified by a mainline list pointer. (The keys are accessible only by DBPL/I statements).

A "list pointer" is a PL/I pointer variable declared in the mainline, set by a DBPL/I GET SET statement or LIST function reference, and used to identify a list. A list pointer having the PL/I NULL pointer value identifies a null (empty) list.

Under OS, 'main storage' for key lists consists of a large randomly accessible file. The list pointer addresses a control block, held in real memory, which describes the list.

There are several ways to form lists (see Figure 1):

1. Read anchor records sequentially and pick keys,

2. Read subrecords sequentially from a subfile and pick keys,

3. Copy an index record cross reference list.

4. Copy a subfile control field.

5. Merge the subfile control fields from a series of anchor records specified in a list,

6. Merge the parent keys from a series of subrecords specified in a list,

7. Get keys sequentially from a list and pick interesting ones,

8. Drop the duplicate keys from a list,

9. Get internal keys sequentially from a list and generate internal keys for an output list,

10. Logically combine (AND, OR, or AND NOT) compatible lists.

The number of keys in a list may be found. Key elements (in external or internal form) may be taken from a list. A list may be used to control READing of anchor records. The mainline may

request and get control of any errors in the use of lists.

Method 1: forming a list of anchor keys:

```
   ptr=NULL;
-->DB((READ FILE(plex) file-positioning;))
|  DB((GET FILE(plex) KEY SET(ptr);))
----------
```

the GET KEY SET may or may not be executed depending on the result of GET FIELD statements, etc.

Method 2: forming a list of subrecord keys:

```
   ptr=NULL;
-->DB((READ FILE(plex) SUBFILE(scfn)
|                 file-positioning;))
|  DB((GET FILE(plex)  SUBFILE(scfn)
|                   KEY SET(ptr);))
----------
```

It is analogous to method 1.

Method 3:

```
DB((READ FILE(plex) INDEX(ifn)
     file-positioning;))
DB((GET FILE(plex) INDEX(ifn)
     LIST SET(ptr);))
```

It may be used on any index.

Method 4:

```
DB((READ FILE(plex) file-positioning;))
DB((GET FILE(plex) SUBFILE(scfn) LIST
SET(ptr);))
```

It copies the multi-element control field as a list of those subrecords in a subfile that are dependent on a particular anchor record, i.e. a "chain" of related detail records. Note that a control field is essentially a stored copy of the result of a whole-subfile search for a particular parent key value.

Method 5:

```
ptr2=CCLIST(plex,scfn,ptr1);
```

It is like method 4 repeated for all the keys in a

Index list with the results all ORed together; It produces a Complete Children List.

Method 6:

```
ptr2=CPLIST(plex,ptr1);
or
ptr2=UPLIST(plex,ptr1);
```

It reads all the subrecords in a list getting the parent key field from each one and merging the parent keys into the output list. The Unique Parent List function drops duplicate parent keys; Complete Parent List does not.

Method 7:

```
   ptr2=NULL;
-->DB((GET LIST(ptr1) KEY <(n)> INTO (var);))
|  DB((GET LIST(ptr1) KEY SET(ptr2);))
----------
```

Where the GET KEY SET may or may not be executed depending on the "var" value. Method 7 essentially handles a special case of method 1 when the "file-positioning" would be governed by a given list and only the key field would be gotten to determine selection; for such a case, method 7 is far more efficient because no record level data base I/O is needed.

Method 8:

```
ptr2 = ULIST(ptr1);
```

It efficiently produces a new list of unique keys (no duplicates) without any record level data base I/O.

Method 9:

```
    DB((SET LIST(ptr2) SIZE(dim)
              LIKE LIST(ptr1);))
--->DB((GET LIST(ptr1) INTERNAL KEY
|                      INTO(var);))
|-->DB((PUT LIST(ptr2) INTERNAL KEY
||      FROM (expr);))
|---------
-----------
```

It is a very special purpose variation of method 7. It works with unconverted external key values. If the inner loop is used, it is possible to

generate more than one key for each GET KEY. Since the output list may receive a multiple or a fraction of the number of keys in the input list, a size dimension must be supplied in the SET LIST LIKE LIST statement estimating the minimum number of output keys.

Method 10:

```
                    '|'
ptr3=LIST(ptr1,'&',ptr2);
                    '-'
```

The LIST function forms a new list in main storage from two compatible lists in main storage. The two argument lists remain accessible for further combination or other use. The LIST function is used in retrieving for compound queries. Given two lists, A and B, the LIST operations provided are illustrated in Figure 2, "Venn Diagram."

When more than two lists have to be combined, the mainline may use one of the following techniques (where R is the resultant intersection list):

```
T1 = LIST (A, '&', B);
T2 = LIST (T1, '&', C);
DB (( FREE LIST (T1))); /*IF DESIRED HERE*/
R  = LIST (T2, '&', D);
DB (( FREE LIST (T2) )); /*IF DESIRED HERE*/
```

A second possible technique is:

```
R = LIST (A, '&', B);
R = LIST (R, '&', C);
R = LIST (R, '&', D);
```

A third possible technique is:

```
R = LIST ( LIST (A, '&', B), '&', LIST (C,
     '&', D));
```

The last two techniques do not retain intermediate lists.

B. List Statements and Functions

#LIST is a PL/I function provided for obtaining the number of keys in a list. For example, if L is a pointer identifying a list and S is a varying-length character string, the following DO-group would be valid:

```
DO I = 1 TO #LIST (L);
     DB (( GET LIST (L) KEY INTO (S); ));
     PUT SKIP LIST (I, S);
     END;
```

If it is merely desired to determine if a list has any elements or not, the following technique is more efficient than a #LIST function reference:

```
IF L ¬= NUIL THEN /* LIST HAS MORE THAN ONE
     ELEMENT */;
```

The GET LIST KEY statement moves a list element key from a list to the user's receiving string. Any conversion from internal to external form is done automatically. The GET LIST INTERNAL KEY statement never converts the list element key value.

The READ statement with the LIST file positioning option is used to read the anchor or subfile record with the next element of a list as its key. It is more efficient than GET LIST KEY; READ by KEY because the internal form of the key element is available for use without conversion.

There are two independent "current elements of a list": the one most recently obtained by a GET LIST KEY statement and the one most recently used by any READ statement under control of the LIST. A key may be referred to sequentially forwards or backwards or by its position (subscript) in a list. The GET or READ current element counter may be reset by a GET LIST KEY(0) or a READ LIST KEY(0) statement respectively.

The SET LIST, LIKE LIST, and PUT LIST INTERNAL KEY statements are for allocating and posting lists for special purposes.

An explicit FREE LIST statement frees the storage and NULLs the pointers for the lists specified. A general FREE LIST statement frees all current lists but does not NULL any pointers.

The ON LISTERROR statement is used to establish a user's list exception routine in the mainline to which the list processing routines return when an exceptional list condition occurs (e.g., attempting to combine incompatible lists). Use of the statement is optional and several ON LISTERROR statements may be executed in a program.

A "list exception routine" must begin with a statement label (the same label identifier specified in an ON statement). PL/I (or DBPL/I) statements may be written following the label to handle the exceptional condition. These statements may reference a binary integer field named LISTERR.ONCODE (declared automatically by the DB preprocessor) for assistance in determining the exceptional condition.

A list exception routine may be terminated in any manner; no provision is made for returning to the function reference that raised the exceptional condition.

VII. RULES AND SYNTACTIC DESCRIPTIONS

The syntax notation used in this manual is a subset of that used in the OS PL/I Reference Manual (Form C28-8201-0) and specified in Section A thereof.

1. A notation variable is shown in lower case letters, hyphens and, possibly, a digit. All such variables shown are defined in this manual either syntactically or semantically.

2. A notation constant denotes the literal occurrence of the characters shown. It consists either of all capital letters or of a special character such as a colon, percent sign, parenthesis, comma or semicolon.

3. Braces, {} , denote that a choice is to be made.

4. Corner brackets, <> , denote options. Anything enclosed in brackets may appear one time or may not appear at all.

5. The vertical stroke, | , indicates that a choice is to be made.

6. An ellipsis, ... , denotes that the contents of the preceding brackets may optionally occur more than once in succession.

'The CCLIST Function'

Complete Children LIST builds a list of subrecord keys from a given parent key list, for a particular subfile, and returns a pointer value identifying the new list to the point of invocation. The new list is the complete list of dependent subrecords (children) formed by merging the parent record's subfile control field lists. Any previously current record and subrecords that were updated will be transmitted to the data base. The record identified by the last (highest) key in the given list will remain as the current (but not locked) record; any current subrecords or index records will remain current. The READ cursor of the given list will be reset.

Reference:

CCLIST (file-name, ctlfield, parent-list-pointer)

A CCLIST function reference is used as or in an expression; it is not to be a subargument in a DB preprocessor function reference. The user may not declare any attributes for the CCLIST function; the following statement will be generated automatically:

DECLARE CCLIST ENTRY (,CHAR(8),PTR) RETURNS (PTR);

Arguments:

file-name: specifies the data base file from which parent records are to be transmitted. It may not be an OUTPUT file. If the file is not open, it will be opened automatically. The 'file-name' must be used in at least one DBPL/I statement elsewhere in the program.

ctlfield: is an expression that specifies the name of the subfile control field. The value of the expression is converted to a character string, if necessary, the first eight characters of which identify the control field.

parent-list-pointer: must be a pointer expression that identifies a list in main storage of parent keys from the data base accessed by 'file-name'. It must have been set when the CCLIST function is invoked.

Result:

The value returned by the CCLIST function is a pointer identifying the new complete children list. The new list will be in order of ascending internal subrecord key values without duplicated values. If none of the parent records have any dependent subrecords in the subfile, a NULL pointer value will be returned.

'The CLOSE Statement'

The CLOSE statement closes a file by disassociating a file name from the self-describing data set with which it was associated by an OPEN. It may also specify that the file be erased.

General Format:

```
CLOSE FILE (file-name) <ERASE>  <,FILE(file-name)
<ERASE>>...;
```

Syntax Rules:

1. The CLOSE statement must be a subargument in a DB preprocessor function reference.

2. Several files can be closed by one CLOSE statement.

General Rules:

1. A closed file can be reopened.

2. Closing an unopened file, or an already closed file, has no effect unless ERASE is specified.

3. If a file is not closed by a CLOSE statement, it is automatically closed at the completion of the program in which it was opened.

4. If the current record and/or subrecords were LOCATEd or updated, closing will cause them to be transmitted to the data base, unlocked (if locked), and disestablished as the current record(s) of the file.

5. The ERASE specification causes the file to be erased and uncatalogued. If the file is a descriptor file, the descriptor region is erased. If the file is an anchor file, the whole data base but not its descriptors is erased. If the file is an associated file, a subfile or an index file, it is erased independently. ERASE is only valid for an UPDATE file.

'The CPLIST Function'

Complete Parent LIST builds a complete list of parent record keys from a given subrecord (children) key list and returns a pointer value identifying the new list to the point of invocation. The new list has the same number of parent keys as the number of subrecord ID keys in the given list. Parent keys will be repeated if more than one of the given subrecord keys are dependent on a particular parent record. The subrecord identified by the last (highest) key in the given list will remain as the current (but not locked) subrecord of that subfile; any current or index records or subrecords of other subfiles will remain current. The READ cursor of the given list will be reset.

Reference:

CPLIST (file-name, child-list-pointer)

A CPLIST function reference is used as or in an expression; it is not to be a subargument in a DE preprocessor function reference. The user may not declare any attributes for the CPLIST function; the following statement will be generated automatically:

```
DECLARE CPLIST ENTRY(,PTR) RETURNS(PTR);
```

Arguments:

file-name:specifies the data base file from which subrecords are to be transmitted. It may not be an OUTPUT file. If the file is not open, it will be opened automatically. The file-name must be used in at least one DBPL/I statement elsewhere in the program.

child-list-pointer: must be a pointer expression that identifies a list in main storage of subrecord keys from the data base accessed by file-name. It must have been set when the CPLIST function is invoked.

Result:

The value returned by the CPLIST function is a pointer identifying the new complete parent list. The new list will be in order of ascending internal parent key values and may have repeated values. If the given subrecord list is null, a NULL pointer value will be returned.

'The DB Preprocessor Function'

DB analyzes a DBPL/I data base access statement during compilation and generates, in its place, suitable PL/I statements for communication with DBPAC. Diagnostic comments may also be generated.

Reference:

<label:>. . . DB ((<<label: ... subargument > ...))

1. One % INCLUDE (DB) preprocessor statement must have been executed by the PI/I compiler before any DB preprocessor function reference is made in a compilation.

2. Several DB preprocessor function references may be made in a compilation.

3. A DB preprocessor function reference may be made only between PL/I statements.

4. When a single DBPI/I statement is to be used as the THEN-unit or ELSE-unit of a PL/I IF statement, the unit must be a PI/I DO-END group enclosing the DB preprocessor function reference.

5. One or more label prefixes may precede a DB preprocessor function reference. They will identify the first executable statement generated for the first subargument.

6. One FINISH statement must be executed by the PL/I compiler as the last subargument of the last DB preprocessor function reference after all other DB preprocessor function references in a compilation.

Argument:

1. The argument of a DB preprocessor function reference is a character string delimited by double enclosing parentheses. Several subarguments can appear in the argument. Each must be a data base access statement having its own terminating semicolon. Blanks and comments may be used freely, as in PL/I, but no PL/I statements are permitted.

2. One or more label prefixes may precede a subargument. They will identify the first executable statement generated for the

subargument.

'The DUPLIST Function'

DUPLIST builds, in dynamically allocated main storage, a compressed copy of a list of keys and returns a pointer value identifying the new list to the point of invocation.

Reference:

```
DUPLIST(list-pointer)
```

A DUPLIST function reference is used as or in an expression; it is not to be a subargument in a DB preprocessor function reference. The user may not declare any attributes for the DUPLIST function; the following statement will be generated automatically:

```
DECLARE DUPLIST ENTRY(POINTER) RETURNS(POINTER);
```

Argument:

list-pointer: must be a pointer expression that identifies a list of keys in main storage. It must have been set when the DUPLIST function is invoked.

Result:

The value returned by the DUPLIST function is a pointer identifying the compressed list copy. A compressed list has none or more list segments of maximum size followed by the last or only list segment allocated to exact length for the remaining keys and all segments are exactly filled; thus, it occupies the least possible main storage.

'The #FIELD Function'

#FIELD calculates the number of elements in a field in the current record or subrecord of a file and returns it to the point of invocation.

Reference:

```
#FIELD (file-name, field-name)
```

A #FIELD function reference is used as or in an expression; it is not to be a subargument in a DB preprocessor function reference. The user may not declare any attributes for the #FIELD function; the following statement will be generated automatically:

```
DECLARE #FIELD ENTRY (,CHARACTER (8)) RETURNS (FIXED
BIN(31);
```

Arguments:

file-name: identifies a data base file. It may not be an OUTPUT file. A current record or subrecord of the file or a subfile must have been established by a DBPL/I READ statement when the #FIELD function is executed. Several #FIELD function references may be executed on a current (sub)record of a file.

field-name: is an expression that specifies the name of the data base field to be examined. The value of the expression is converted to a character string, if necessary, the first eight characters of which identify the field. Any field may be examined.

Result:

The value returned by the #FIELD function is a binary integer of maximum precision giving the number of elements in the field in the current (sub)record of the file. If the field has a null value, a zero value will be returned.

'The FINISH Statement'

The FINISH statement causes the DB preprocessor to complete its analysis of all data base access statements and its generation of suitable PL/I statements. A RETURN Statement will be generated which will terminate execution of the procedure. A CLOSE is also generated for those file-names utilized by the compilation. Diagnostic comments may also be generated.

General Format:

```
FINISH;
```

Syntax Rule:

One FINISH statement must be used after all other data base access statements in a compilation. It must be the last subargument in a DB preprocessor function reference.

'The FREE LIST Statement'

The FREE LIST statement frees main storage previously dynamically-allocated for one or more lists of (cross-reference) keys.

General Format:

```
FREE LIST <(list-pointer<,list-pointer> ...)>;
```

Syntax Rules:

1. The FREE LIST statement must be a subargument in a DB preprocessor function reference.

2. Several lists may be explicitly freed by one FREE LIST statement.

General Rules:

1. If a list-pointer is explicitly specified, it must be a pointer expression that identifies a list of keys in main storage. It must have been set when the FREE LIST statement is executed.

   a. If the value of the list-pointer is NULL, no action will be taken for that list pointer.

   b. If the value of the list-pointer is not NULL, the dynamic main storage for the list of keys identified by it will be freed and the list-pointer will be given a NULL pointer value.

   c. FREE LIST will ignore a pointer which addresses a list which has been posted in SETAB and assigned a set number. No action will be taken.

2. If no list-pointer is explicitly specified in the FREE LIST statement, all dynamic list storage will be freed. The user's list pointers are not given NULL values; it is the user's responsibility not to use them for list identification until they are reset. If no dynamic list storage has been previously allocated, this option of the FREE LIST statement will have no effect.

'The GET FIELD Statement'

The GET FIELD statement moves a data element value from the current record or subrecord of a file to the user's receiving field; it may cause the value to be converted from an internal form to a display form.

General Format:

```
GET FILE (file-name) FIELD  (field-name <, field-name> ... )
                     INTO   (variable <, variable 2 ... );
```

Syntax Rules:

1. The GET FIELD statement must be subargument in a DB preprocessor function reference.

2. Several data element values can be moved by one GET FIELD statement. In this case, a corresponding variable must be specified for each field name.

General Rules:

1. The data element value will be taken from the data base file specified in the FILE clause. It may not be an OUTPUT file.

2. A current record or subrecord of the file or a subfile must have been established by a READ statement when the GET statement is executed. Several GET FIELD statements may be executed on a current (sub)record of the file.

3. The field-name is an expression that specifies the name of the data base field from which the data element value is to be obtained. The value of the expression is converted to a character string, if necessary, the first eight characters of which identify the field. If the user who executes the GET FIELD statement is not the owner of the file, the field-name may not specify a field that the owner has not authorized the user to GET.

   a. If the field is not subdivided into elements, the data element value (possibly null) will be taken from the field in the current record of the file.

   b. If the field is a multiple-element field, the data element value will be taken from the next element of the field, in left to right order, following the element taken by the

previous GET of the FIELD of the current record of the file. If there has been no previous GET of the FIELD since the record was READ, the leftmost element is taken unless the field is null, in which case, a null element value will be generated. If a previous GET of a FIELD since the record was READ took the last (rightmost) element, a null value will be generated.

4. The variable in the INTO clause specifies the user's receiving field. It must be the identifier of a varying length character string variable declared by the user. The internal form of the data element value will be taken as a varying length character string (of zero length, if the value is null), converted to display form and assigned to the variable. If the length of the display form of the value exceeds the user-declared maximum length of the variable, the value will be truncated and an error condition raised.

'The GET INDEX KEY Statement'

The GET INDEX KEY statement moves the key value from the current record of an index to the user's receiving field; it may cause the value to be converted from an internal form to a display form.

General Format:

GET FILE (file-name) INDEX (indfield) KEY INTO (variable);

Syntax Rule:

The GET INDEX KEY statement must be a subargument in a DB preprocessor function reference.

General Rules:

1. The FILE clause specifies the data base file from which an index key value is to be taken. It may not be an OUTPUT file.

2. The INDEX clause specifies the index file from which the current index key value is to be taken. The indfield expression value is converted to a character string, if necessary, the first eight characters of which identify the indexed field.

3. A current record of the index must have been established by a READ INDEX statement when the GET INDEX KEY statement is executed.

4. The variable in the INTO clause specifies the user's receiving field. It must be the identifier of a varying length character string variable declared by the user. The internal form of the index key value will be taken as a varying length character string, converted to display form and assigned to the variable. If the length of the display form of the value exceeds the user-declared maximum length of the variable, the value will be truncated and an error condition raised.

'The GET KEY SET Statement'

The GET KEY SET statement moves the internal key value from a current record or subrecord of a file to a list of keys in dynamically allocated main storage and sets a pointer identifying the list or extends an existent list.

General Format:

GET FILE (file-name) <SUBFILE (ctlfield)> KEY SET (list-pointer);

Syntax Rule:

The GET KEY SET statement must be a subargument in a DB preprocessor function reference.

General Rules:

1. The FILE clause specifies the data base file from which a key value is to be taken. It may not be an OUTPUT file.

2a. If no SUBFILE clause is present, the internal key value will be taken from the current root record.

b. A SUBFILE clause specifies that the internal key value from a current subrecord is to be taken. The ctlfield expression value is converted to a character string, if necessary, the first eight characters of which identify the control field.

3. A current (sub)record must have been established by a READ or READ SUBFILE statement when the GET KEY SET statement is executed.

4. The list-pointer in the SET clause specifies the user's pointer identifying the list of keys in main storage. It must be the identifier of a pointer variable declared by the user.

4a. If the user assigns the NULL value to his list-pointer before executing the GET KEY SET statement, main storage will be dynamically allocated automatically for a new list, the key value will be moved there from the current (sub)record, and the list-pointer value will be set to identify the list in main storage. The list remains allocated in main storage until the user executes a FREE LIST statement.

b. Otherwise, the list-pointer should identify a list of keys in main storage to which another compatible key value is to be appended. It must have been set (by the user assigning NULL and executing a GET KEY SET statement as described above) when this GET KEY SET statement is executed. The key value will be moved from the current (sub)record. The list-pointer will be unchanged.

'The GET LIST INTERNAL KEY INTO Statement'

The GET LIST INTERNAL KEY INTO statement increments the internal GET cursor of a list of keys in main storage identified by a list pointer and moves the indicated key value in internal form to the user's receiving field.

General Format:

GET LIST (list pointer) INTERNAL KEY INTO (variable);

Syntax Rule:

The GET LIST INTERNAL KEY INTO statement must be a subargument in a DB preprocessor function reference.

General Rules:

1. The list-pointer must be a pointer expression that identifies a list of keys in main storage from which the next key value is to be taken. It must have been set when the GET LIST INTERNAL KEY INTO statement is executed. In the exceptional case of a list pointer having a NULL pointer value, a null string value will be generated.

2. The internal GET cursor of the list will be incremented to indicate that the next element of the list, in order of ascending internal key values, is current and will be taken. (If the internal GET cursor was reset, the element having the lowest internal key value is current and will be taken. If the internal GET cursor was on the last element (highest internal key value), the cursor will be reset and a null string value will be generated.)

3. The variable in the INTO clause specifies the user's receiving field. It must be the identifier of a varying length character string variable declared by the user. The internal form of the key value will be taken as a varying length character string (cf zero length on end of list) and assigned without formatting to the variable. If the length of the internal form of the value exceeds the user-declared maximum length of the variable, the value will be truncated and an error condition raised.

'The GET LIST KEY(0) Statement'

The GET LIST KEY(0) statement resets the internal GET cursor of a list of keys in main storage.

General Format:

GET LIST (list-pointer) KEY(0);

Syntax Rule:

The GET LIST KEY(0) statement must be a subargument in a DB preprocessor function reference.

General Rules:

1. The list-pointer must be a pointer expression that identifies a list of keys in main storage whose GET cursor is to be reset. The list-pointer must have been set when the GET LIST KEY(0) statement is executed. In the exceptional case of a list-pointer having a NULL pointer value, no action will occur and no error condition will be raised.

2. The internal GET cursor of the list will be reset (as it was when the list was built).

'The GET LIST KEY INTO Statement'

The GET LIST KEY INTO statement increments or sets the internal GET cursor of a list of keys in main storage identified by a list pointer and moves the indicated key value to the user's receiving field; it may cause the value to be converted from internal to display form.

General Format:

GET LIST (list-pointer) KEY <(rel-key)> INTO (variable);

Syntax Rule:

The GET LIST KEY INTO statement must be a subargument in a DB preprocessor function reference.

General Rules:

1. The list-pointer must be a pointer expression that identifies a list of keys in main storage from which the key value is to be taken. It must have been set when the GET LIST KEY INTO statement is executed. In the exceptional case of a list pointer having a NULL pointer value, a null string value will be generated.

2a. If no rel-key is specified, the internal GET cursor of the list will be incremented to indicate that the next element of the list, in order of ascending internal key values, is current and will be taken. (If the internal GET cursor was reset, the element having the lowest internal key value is current and will be taken. If the internal GET cursor was on the last element the cursor will be reset and a null string value will be generated.)

b. If a rel-key expression is specified, its value will be converted, if necessary, to a fixed binary integer of maximum precision.

If rel-key has a negative value, such as -1, the internal GET cursor of the list will be decremented to indicate that the previous element of the list, in order of internal key values, is current and will be taken. (If the internal GET cursor was reset, the element having the highest internal key value is current and will be taken. If the internal GET cursor was on the first element, the cursor will be reset and a null string value will be generated.)

If rel-key has a positive value, the internal GET

cursor of the list will be set to indicate that rel-key the relative element of the list is current and will be taken. (If rel-key is zero or greater than the number of keys in the list, the cursor will be reset and a null string value will be generated.)

3. The variable in the INTO clause specifies the user's receiving field. It must be the identifier of a varying length character string variable declared by the user. The internal form of the key value will be taken as a varying length character string converted to display form and assigned to the variable. If the length of the display form of the value exceeds the user-declared maximum length of the variable, the value will be truncated and an error condition raised.

'The GET LIST KEY SET Statement'

The GET LIST KEY SET statement moves the current internal key value from a list of keys identified by a list pointer to a new list in dynamically allocated main storage and sets a pointer identifying the new list or extends an existent list.

General Format:

GET LIST (list-pointer) KEY SET (new-list-pointer);

Syntax Rule:

The GET LIST KEY SET statement must be a subargument in a DB preprocessor function reference.

General Rules:

1. The list-pointer must be a pointer expression that identifies a list of keys in main storage having a non-zero GET cursor indicating a current key.

2. The internal key value will be taken from the current element of the list indicated by the internal GET cursor. The GET cursor will be unchanged.

3. The new-list-pointer in the SET clause specifies the user's pointer identifying the new list of keys in main storage. It must be the identifier of a pointer variable declared by the user.

3a. If the user assigns the NULL value to his new-list-pointer before executing the GET LIST KEY SET statement, main storage will be dynamically allocated automatically for a new list, the key value will be moved there, and the new-list-pointer value will be set to identify the new list in main storage. The new list remains allocated in main storage until the user executes a FREE LIST statement.

3b. Otherwise, the new-list-pointer should identify a list of keys in main storage to which another compatible key value is to be appended. It must have been set when this GET LIST KEY SET statement is executed. The key value will be moved and the new-list-pointer will be unchanged.

'The GET LIST SET Statement'

The GET LIST SET statement moves a list of keys from the current record of an index or from a subfile control field in the current root record to dynamically allocated main storage and sets a pointer identifying it.

General Format:

```
GET FILE(filename) <INDEX(indfield)>LIST SET (list-pointer)
                   <SUBFILE(ctlfield)>
```

Syntax Rule:

The GET LIST SET statement must be a subargument in a DB preprocessor function reference.

General Rules:

1. The FILE clause specifies the data base file from which the list of keys is to be taken. It may not be an OUTPUT file.

2a. If an INDEX clause is specified, a current index record must have been established by a READ INDEX statement when the GET INDEX LIST SET statement is executed. The INDEX clause specifies the index file from which the list of (cross-reference) keys is to be taken. The indfield expression value is converted to a character string, if necessary, the first eight characters of which identify the indexed field.

2b. If a SUBFILE clause is specified, a current root record must have been established by a READ statement when the GET LIST SET statement is executed. The ctlfield expression value is converted to a character string, if necessary, the first eight characters of which identify the control field from which the list of keys (children) is to be taken. If the user who executes the GET SUBFILE LIST SET statement is not the owner of the file, the ctlfield may not specify a control field that the owner has not authorized the user to GET.

2c. If neither an INDEX nor a SUBFILE clause is specified, the FILE must be an INPUT file opened with a TITLE for independent access to a particular inverted index file and a current record must have been established by a READ statement when the GET LIST SET statement is executed. The list of (cross-reference) keys will

be taken.

3. The list-pointer in the SET clause specifies the user's pointer to be set to identify the list of keys in main storage. It must be the identifier of a pointer variable declared by the user.

   a. If the list of keys field of the current record is null, the list-pointer will be given a NULL pointer value. (This occurs for the SUBFILE case when the control field is null indicating no subordinate (children) subrecords.)

   b. Otherwise, main storage will be dynamically allocated automatically for the list, the list of keys will be moved there from the current record, and the list-pointer value will be set to identify the list in main storage. The list remains allocated in main storage until the user executes a FREE LIST statement.

'The GET RECORD Statement'

The GET RECORD statement moves a physical record in internal form from the current record of a file to the user's receiving field.

General Format:

```
GET FILE (file-name) RECORD INTO (variable);
```

Syntax Rule:

The GET RECORD statement must be a subargument in a DB preprocessor function reference.

General Rules:

1. The physical record will be taken from the current record of the file specified in the FILE clause. It must be an UPDATE or INPUT file owned by the user who executes the GET RECORD statement.

2. A current record of the file must have been established by a READ statement when the GET statement is executed. Several GET statements may be executed on a current record of the file.

3. The variable in the INTO clause specifies the user's receiving field. It must be the identifier of a structure or fixed-length character string variable declared by the user. The internal self-defining physical record will be moved into the variable without any conversion. No receiving field length checking will be done. (A GET FIELD 'RECLEN' statement may be used for this purpose.)

'The % INCLUDE LISRMAC (DB) Preprocessor Statement'

The % INCLUDE LISRMAC (DB) preprocessor statement causes the text of the DB preprocessor function to be taken from the system source library during compilation, incorporated in the source program and activated.

General Format:

```
% INCLUDE LISRMAC (DB);
```

Syntax Rule:

Only one % INCLUDE DB preprocessor statement may be used in the source text for a compilation. It must immediately follow the beginning PROCEDURE statement, before any other statements, if the compilation contains DB preprocessor function references for data base access statements.

'The LIST Function'

LIST derives a new list of (cross-reference) keys from two given lists of keys and returns a pointer value identifying the new list to the point of invocation. The new list may be the union or intersection of the given lists or the sublist of the first given list excluding the second.

Reference:

LIST (list-pointer-1, operator, list-pointer-2)

A LIST function reference is used as or in an expression; it is not to be a subargument in a DB preprocessor function reference. The user may not declare any attributes for the LIST function; the following statement will be generated automatically:

```
DECLARE LIST ENTRY (POINTER, CHARACTER(1), POINTER)
     RETURNS(POINTER);
```

Arguments:

Each of the two list-pointer arguments must be a pointer expression that identifies a list of keys in main storage. Each must have been set when the LIST function is invoked. The lists of keys identified must be compatible (having the same internal key element length, etc.).

The operator argument is an expression that specifies the list operation to derive the new list. The value of the operator will be converted, if necessary, to a one-character string. The value must be:

logical OR, '|', specifying the union,

logical AND, '&', specifying the intersection, or

minus sign, '-', specifying the sublist of the first list excluding the second list.

Result:

The value returned by the LIST function is a pointer identifying the new list. The new list will be in order of ascending internal key values without duplicated key values (unless there are duplicates in one of the argument lists). If the new list is null, the value returned may be assigned to one of the argument list pointers; however, the argument list would then be lost to the mainline (unless the user had assigned its pointer value to another pointer previously) and could not be explicitly freed (but FREE LIST; would free it and all other lists).

'The #LIST Function'

#LIST calculates the number of (cross-reference) keys in a list of keys identified by a list pointer and returns it to the point of invocation.

Reference:

```
#LIST (list-pointer)
```

A #IIST function reference is used as or in an expression; it is not to be a subargument in a DB preprocessor function reference. The user may not declare any attributes for the #LIST function; the following statement will be generated automatically:

```
DECLARE #LIST ENTRY (POINTER) RETURNS (FIXED
BINARY(31));
```

Argument:

The list-pointer argument must be a pointer expression that identifies a list of keys in main storage. It must have been set when the #LIST function is invoked.

Result:

The value returned by the #LIST function is a binary integer of maximum precision giving the number of keys in the list identified by the list-pointer argument. If the list-pointer has a NULL pointer value, a zero value will be returned.

'The LOCATE Statement'

The locate statement, which applies to OUTPUT or DIRECT UPDATE files, causes formation of a new current record having a key field and having all other fields null; it may also cause transmission of the previously current record to the data base.

General Format:

LOCATE FILE (file-name) KEYFROM (expression);

Syntax Rule:

The LOCATE statement must be a subargument in DB preprocessor function reference.

General Rules:

1. The FILE clause specifies the data base file to which the record is to be subsequently transmitted. It must be owned by the user who executes the LOCATE statement. It may not be an INPUT or SEQUENTIAL UPDATE file.

2. If the file is not open, it is opened automatically.

3. The value of the expression in the KEYFROM clause is converted to a varying length character string, if necessary, validated and/or converted to an internal form.

   a. If the file has the SEQUENTIAL OUTPUT attributes, the internal key is checked for ascending sequence and subsequently used as the key of the record when it is transmitted to the data base.

   b. If the file has the DIRECT attribute, a READ KEY is attempted using the internal key. If the key is found, a duplicate key error condition is raised and the LOCATE statement has the effect of the READ KEY statement. If the key is not found, it is subsequently used as the key of the record when it is transmitted to the data base.

4. After execution of the LOCATE statement, subrecords may be LOCATEd and values may be PUT into fields (other than the key) of the record for subsequent transmission to the data base, which will occur immediately before the next LOCATE,

READ, CLOSE or automatic close operation on the file.

'The LOCATE SUBFILE Statement'

The LOCATE SUBFILE statement causes formation of a new current subrecord having a key field and a parent keyfield and having all other fields null; it also causes the new key to be automatically entered in the parent record control field; it may also cause transmission of the previously current subrecord of the subfile.

General Format:

```
LOCATE FILE (file-name) SUBFILE (ctlfield);
```

Syntax Rule:

The LOCATE SUBFILE statement must be a subargument in a DB preprocessor function reference.

General Rules:

1. The FILE clause specifies the data base file to which the subrecord is to be subsequently transmitted. It must be owned by the user who executes the LOCATE SUBFILE statement. It may not be an INPUT file.

2. A current record of the file must have been established when the LOCATE SUBFILE statement is executed. Several LOCATE SUBFILE statements for one or more subfiles may be executed on a current record of the file.

3. The ctlfield is an expression that specifies the name of the subfile control field. The value of the expression is converted to a character string, if necessary, the first eight characters of which identify the control field.

4. After execution of the LOCATE SUBFILE statement, values may be PUT into fields of the subrecord for subsequent transmission to the data base, which will occur immediately before the next LOCATE SUBFILE or READ SUBFILE on this subfile or before the next CLOSE or automatic close on the file.

'The ON Statement'

The ON statement specifies what action is to be taken when an interruption results from the occurrence of the specified error condition.

General Format:

```
ON  <ERRORFILE(file-name)>  <SYSTEM     > ;
    <LISTERROR            >  <GO TO label>
```

Syntax Rule:

The ON statement must be a subargument in a DB preprocessor function reference.

General Rules:

1. The ON statement determines how an error occurring for the specified condition is to be handled. Whether the error is handled in the standard DB fashion or by a user-supplied method is determined by the action specification in the ON statement, as follows:

   a. If the action specification is SYSTEM, the standard DB action is taken. For most conditions, the system simply posts the ONCODE field and raises the ERROR condition. (Note that the standard DB action is always taken if an interruption occurs and no ON statement for the condition is in effect.)

   b. If the action specification is GO TO, the user has supplied his own error-handling action at label. Control is not transferred to label when the ON statement is executed; control is transferred only when an error results from the occurrence of the specified condition.

2. The action specification established by executing an ON statement remains in effect unless it is over-ridden by the execution of another ON statement specifying an action for the same condition.

'The OPEN Statement'

The OPEN statement opens a file by associating a file name with a DATA BASE. It may also specify attributes for the file.

General Format:

```
OPEN FILE (file-name) <TITLE (expression)> <access>
<function>
 <,FILE(file-name) <TITLE(expression)> <access>
<function>>...;

where "access" is:
  DIRECT | SEQUENTIAL

and "function" is:
  INPUT | OUTPUT | UPDATE
```

Syntax Rules:

1. The OPEN statement must be a subargument in a DB preprocessor function reference.

2. Several files can be opened by one OPEN statement.

General Rules:

1. If a file is not opened by an OPEN statement, it is automatically opened when a READ or LOCATE statement for the file is first executed.

2. Opening an already opened file by an OPEN statement causes it to be closed and reopened.

3. If the TITLE option is specified, the value of the expression is converted to a character string the first eight characters of which identify the data base to be associated with the file. If the TITLE option is not specified, the file-name is taken to identify the data base.

4. If no access attribute is specified, DIRECT is the default unless a WRITE statement on the file is used in the same compilation, thus implying the SEQUENTIAL attribute.

5. If a function attribute is specified, it determines the direction of data transmission permitted for the file. If no function attribute is specified, it is implied from the usage of other data base statements on the same file in the

compilation (e.g., REPUT implies UPDATE). If no other data base statements on the same file appear in the compilation, the default is INPUT. The only user permitted to access and OUTPUT or UPDATE file is the owner of that file.

'The PUT FIELD Statement'

The PUT FIELD statement moves a data element value to the current record or subrecord of a file for subsequent transmission to the data base; it may cause the value to be validated and/or converted to an internal form and it may also cause a cross-reference to be automatically entered in an index file.

General Format:

```
PUT FILE (file-name) FIELD (field-name<,field-name> ...)
            FROM (expression<, expression> ...);
```

Syntax Rules:

1. The PUT FIELD statement must be a subargument in a DB preprocessor function reference. 'The READ Statement'
2. Several data element values can be moved by one PUT FIELD statement. In this case, a corresponding expression must be specified for each field-name.

General Rules:

1. The FILE clause specifies the data base file to which the data element value is to be subsequently transmitted. It must be an OUTPUT or UPDATE file owned by the user who executes the PUT statement. It may not be an associated file or an index file.

2. A current exclusive record or subrecord (depending on the field-name) of the file or subfile must have been established when the PUT statement is executed. Several PUT statements may be executed on a current exclusive (sub)record of the file.

3. The field-name is an expression that specifies the name of the data base field into which the data element value is to be moved. The value of the expression is converted to a character string the first eight characters of which identify the field. The field-name may not specify the key field of the record or any other read only field. The PUT statement moves a value to a field element that had no previous value.

   a. If the field is not subdivided into elements, it must have had a null value before the PUT statement is executed to give it a value.

b. If the field is a multiple-element field, a new element will be added at the right end of the field.

4. The expression in the FROM clause specifies the data value to be given to the field element. The value of the expression is converted to a varying length character string, if necessary, validated and/or converted to an internal form and moved into the current record of the file. (If the data base field element is variable-length, other fields are automatically shifted to make room.) The varying length character string value after any conversion to an internal form must have a length greater than zero; i.e., a null string is an invalid data value for a PUT statement.

5. If the data base field has an inverted index file, a cross-reference of the internal data element value to the key of the (sub)record will be automatically entered in the inverted index file.

6. The (sub)record with the new data element value will be transmitted to the data base when an UNLOCK statement for the (sub)file is executed or immediately before the next LOCATE or READ on the (sub)file or immediately before the next CLOSE or automatic close operation on the file.

'The PUT LIST INTERNAL KEY FROM Statement'

The PUT LIST INTERNAL KEY FROM statement moves an internal key value to extend a list of keys in main storage.

General Format:

```
PUT LIST (list-pointer) INTERNAL KEY FROM (expression
                                  <,expression>...);
```

Syntax Rules:

1. The PUT LIST INTERNAL KEY FROM statement must be a subargument in a DB preprocessor function reference.

2. Several internal key values can be moved by one PUT LIST INTERNAL KEY FROM statement.

General Rules:

1. The list-pointer in the LIST clause specifies the user's pointer identifying the list of keys in main storage to which the internal key value is to be moved. It must have been set when the PUT LIST INTERNAL KEY FROM statement is executed. In the case of a list pointer having a NULL pointer value, a list error condition will be raised.

2. The expression in the FROM clause specifies the internal key value to be moved to the list. The value of the expression is converted to a varying length character string which must be the same length as the list element size. If the length is different or zero (null) an error condition will be raised.

'The READ Statement'

The READ statement causes a parent record or a subrecord to be transmitted from the data base and established as the current record of the file (or as the current subrecord of a subfile); it may also cause transmission of the previously current record (or subrecord of a subfile) to the data base.

*When READing according to a LIST of subrecord ID keys.

General Format:

```
READ FILE (file-name) <file-positioning> <NOLOCK>;
```

where file-positioning may be:

```
KEY(expression) |
 LIST(list-pointer) <KEY (rel-key)> |
 PER SUBFILE (ctlfield)
```

Syntax Rule:

The READ statement must be a subargument in a DB preprocessor function reference.

General Rules:

1. The FILE clause specifies the data base file from which the record is to be transmitted. It may not be an OUTPUT file.

2. If the file is not open, it will be opened automatically unless PER SUBFILE is specified.

3a. If no file positioning option is specified, the next sequential record, following the one previously read, will be transmitted. If the file is newly opened, the record having the lowest internal key value will be transmitted.

b. If the KEY file-positioning option is specified, the value of the expression will be converted to a varying length character string, validated and/or converted to an internal form and used to determine which record will be transmitted. If the key cannot be found, a key error condition will be raised, but the record having the next lower internal key value will be transmitted.

c. If the LIST file-positioning option is specified, the file may not be an index file. The list-pointer must be a pointer expression that

identifies a list of anchor or subrecord keys in main storage to control the READing. It must have been set when the READ statement is executed. The keys in the file list identified must be compatible with the internal anchor keys of the file, or with the subrecord keys of one of its subfiles. In the latter case the list determines which subfile will be accessed for a subrecord to be made current. In the case of a list-pointer having a NULL pointer value, a key error condition will be raised and no record will be transmitted.

If the LIST clause is not followed by a KEY clause, the internal READ cursor of the list will be incremented to indicate that the next element of the list, in order of ascending internal key values, will be used to determine which (sub)record will be transmitted. (If the internal READ cursor was reset, the element having the lowest internal key value will be used. If the internal READ cursor was on the last element, the cursor will be reset, a key error condition will be raised, and no (sub)record will be transmitted.)

If the LIST clause is followed by a KEY clause, the value of the rel-key expression will be converted to a fixed binary integer of maximum precision.

If rel-key has a value of zero, the internal READ cursor of the list will be reset. No (sub)record will be transmitted and no error condition will be raised.

If rel-key has a negative value, such as -1, the internal READ cursor of the list will be decremented to indicate that the previous element of the list, in order of internal key values, will be used to determine which (sub)record will be transmitted. (If the internal READ cursor was reset, the element having the highest internal key value will be used. If the internal READ cursor was on the first element the cursor will be reset, a key error condition will be raised, and no (sub)record will be transmitted.)

If rel-key has a positive value the internal READ cursor of the list will be set to indicate that the element in the rel-key position of the list will be used to determine which (sub)record will be transmitted. (If rel-key is greater than the

number of keys in the list, the cursor will be reset, a key error condition will be raised, and no (sub)record will be transmitted.)

d. If the PER SUBFILE file-positioning option is specified, the parent record of a current subrecord will be transmitted. The value of the ctlfield expression will be converted to a character string the first eight characters of which identify the subfile control field. A current subrecord of the subfile must have been established by a READ SUBFILE statement when the READ PER SUBFILE statement is executed. The internal parent key field value on the subrecord will be used to determine which record will be transmitted.

e. No KEYTO option is provided. A GET FIELD statement, following a READ statement, may be used for this purpose.

4. Any READ statement referring to an UPDATE file will cause the record to be locked for exclusive use unless the NOLOCK option is specified. A locked record cannot be READ by any other task until it is unlocked. Any attempt to READ a record locked by another task results in a wait. Subsequent unlocking is accomplished by the locking task through the execution of an UNLOCK, READ, LOCATE, CLOSE or implicit close operation on the file.

'The READ INDEX Statement'

The READ INDEX statement causes an index record to be transmitted from the data base and established as the current record of the index.

General Format:

READ FILE (file-name) INDEX (indfield) <index-positioning>;

where index-positioning may be:

KEY(expression)

Syntax Rule:

The READ INDEX statement must be a subargument in a DB preprocessor function reference.

General Rules:

1. The FILE clause specifies the data base file from which an index record is to be transmitted. It may not be an OUTPUT file.

2. If the file is not open, it will be opened automatically.

3. The INDEX clause specifies the index file from which the index record is to be transmitted. The indfield expression value is converted to a character string, if necessary, the first eight characters of which identify the indexed field. If the user who executes the READ INDEX statement is not the owner of the file, the indfield may not specify a field that the owner has not authorized the user to GET.

4a. If no index-positioning option is specified, the file must be an INPUT file. The next sequential index record, following the one previously read, will be transmitted. If the index file has not been previously read, the record having the lowest indexed field value will be transmitted.

b. If the KEY index-positioning option is specified, the file may be an INPUT or UPDATE file. The value of the expression will be converted to a varying length character string, if necessary, validated and/or converted to an internal index key form and used to determine which index record will be transmitted. If the key cannot be found, a key error will be raised, but the index record

having the next lcwer internal index key value will be transmitted.

c. No KEYTO option is provided. A GET INDEX KEY statement, following a READ INDEX statement, may be used for this purpose.

5. A READ INDEX statement never locks an index record for exclusive use.

'The READ SUBFILE Statement'

The READ SUBFILE statement causes a subrecord to be transmitted from the data base and established as the current subrecord of the subfile.

General Format:

```
READ FILE (file-name) SUBFILE(ctlfield)<subfile-positioning>
                                        <NOLOCK>;
```

where subfile-positioning may be:

KEY(expression)

Syntax Rule:

The READ SUBFILE statement must be a subargument in a DB preprocessor function reference.

General Rules:

1. The FILE clause specifies the data base file from which a subrecord will be transmitted. It may not be an OUTPUT file.

2. If the file is not open, it will be opened automatically.

3. The SUBFILE clause specifies the subfile from which the subrecord is to be transmitted. The ctlfield expression value is converted to a character string, if necessary, the first eight characters of which identify the subfile control field. If the user who executes the READ SUBFILE statement is not the owner of the file, the ctlfield may not specify a subfile that the owner has not authorized the user to READ.

4.a If no subfile-positioning option is specified, the file must be an INPUT file. The next sequential subrecord following the one previously read, will be transmitted. If the subfile has not been previously read, the subrecord having the lowest subrecord ID key value will be transmitted.

4.b If the KEY subfile-positioning option is specified, the file may be an INPUT or UPDATE file. The value of the expression will be converted to a varying length character string, if necessary, converted from numeric character to binary (24, 7) internal subrecord key form and used to determine which subrecord will be

transmitted. If the subrecord key cannot be found, a key error condition will be raised, but the subrecord having the next lower internal subrecord key value will be transmitted.

4.c No LIST subfile-positioning option is provided for the READ SUBFILE statement; the regular READ with LIST file-positioning may be used for this purpose because the list determines if and which subfile is to be accessed.

4.d No subfile-positioning option is provided for reading the region of subrecords dependent on the current root record; GET SUBFILE LIST SET followed by READ LIST statements are very flexible for this purpose.

4.e No KEYTO option is provided. A GET FIELD statement, following a READ statement, may be used for this purpose.

5. A READ SUBFILE statement referring to an UPDATE file will cause the subrecord to be locked for exclusive use unless the NOLOCK option is specified. A locked subrecord cannot be READ by any other task until it is unlocked. Any attempt to READ a subrecord locked by another task results in a wait. Subsequent unlocking is accomplished by the locking task through the execution of an UNLOCK SUBFILE, READ SUBFILE, or LOCATE SUBFILE operation on the subfile or a CLOSE or implicit close operation on the file.

'The REPUT Statement'

The REPUT statement replaces a data element in the current record or subrecord of an UPDATE file for subsequent retransmission to the data base; it may cause the value to be validated and/or converted to an internal form and it may also cause a cross-reference to be automatically deleted and another entered in an index file. The REPUT statement may be used to delete a whole record or subrecord and all cross-references to it in index files.

General Format:

```
REPUT FILE(file-name) FIELD(field-name<, field-name> ...)
                      FROM(expression<, expression> ...);
```

Syntax Rules:

1. The REPUT statement must be a subargument in a DB preprocessor function reference.

2. Several data element values can be replaced by one REPUT statement. In this case, a corresponding expression must be specified for each field-name.

General Rules:

1. The FILE clause specifies the data base file to which the data element value is to be subsequently retransmitted. It must be an UPDATE file owned by the user who executes the REPUT statement. It may not be an associated file or an index file.

2. A current exclusive record of the file must have been established when the REPUT statement is executed. Several REPUT statements may be executed on a current exclusive record of the file.

3. The field-name is an expression that specifies the name of the data base field whose data element value is to be replaced. The value of the expression is converted to a character string the first eight characters of which identify the field.

a. If the field is the key field of an anchor record, the expression in the FROM clause must have a null value (zero length) and the whole root record and all of its dependent subrecords in all subfiles of the FILE will be deleted.

b. If the field is the key field of a subrecord, both the subrecord and its parent record must be current. The expression in the FROM clause must have a null value (zero length) and the whole subrecord will be deleted.

c. Otherwise the field-name may not specify a read-only field.

If the field is not subdivided into elements, its value will be replaced. If the field is a multiple-element field, the element taken by the last GET of the FIELD since the current (sub)record of the file was READ will have its value replaced. If no element was fpimd fpr tje GET FIELD or if no GET cf the FIELD of the current (sub)record of the file was executed, an error condition is raised.

4. The expression in the FROM clause specifies the new data value to be given to the field element. The value of the expression is converted to a varying length character string, validated and/or converted to an internal form and moved into the current (sub)record of the file. (If the data base field element is variable-length and the new value's length is different from the old, other field elements are autcmatically shifted as necessary.)

5a. If the data base field is the key field of the anchor record and the expression in the FROM clause has a null value, all cross-references to the key of the parent reccrd and its dependent subrecords will be automatically deleted from all index files for the file specified in the FILE clause.

b. If the data base field is the ID key field of a subrecord and the expression in the FROM clause has a null value, all crcss-references to the ID key of the subrecord will be automatically deleted from all index files for the subfile.

c. If the data base field has an index file, the cross reference of the old internal data element value will be automatically deleted, and a cross-reference of the new internal data element value to the key of the record will be automatically entered in the index file.

6. The (sub)record with the new data element value will be retransmitted to the data base when an

UNLOCK statement for the (sub)file is executed or immediately before the next LOCATE or READ on the (sub)file or immediately before the next CLOSE or automatic close operation on the file.

'The SET LIST LIKE LIST Statement'

The SET LIST LIKE LIST statement dynamically allocates main storage for a new list to later contain an estimated number of keys, copies the key field name and conversion routine name etc., from an existing list, and sets a pointer identifying the new list.

General Format:

```
SET LIST (new-list-pointer) SIZE (dimension) LIKE LIST
(list-pointer);
```

Syntax Rule:

The SET LIST LIKE LIST statement must be a subargument in a DB preprocessor function reference.

General Rules:

1. The list-pointer in the LIKE LIST clause must be a pointer expression that identifies a list of keys in main storage to be referenced for prefix information such as key element length etc. In the exceptional case of a list pointer having a NULL pointer value, a NULL pointer value will be returned.

2. The SIZE clause specifies an estimate of the number of keys that will subsequently be put into the new list. For example, it could be the #LIST count of the existing list or a multiple of it. The dimension expression value will be converted, if necessary, to a fixed binary integer of maximum precision and used to govern the allocation of the first segment of the new list.

3. The new-list-pointer in the SET LIST clause specifies the user's pointer identifying the new list of keys in main storage. It must be the identifier of a pointer variable declared by the user. Regardless of its former value, it will be set to identify the new list of keys in main storage. The new list remains allocated in main storage until the user executes a FREE LIST statement.

'The ULIST Function'

ULIST builds a copy of a list of keys omitting duplicated key values and returns a pointer value identifying the new list to the point of invocation. If The given list has only unique key values, ULIST returns the given list pointer without copying the list.

Refernece:

ULIST(list-pointer)

A ULIST function reference is used as or in an expression; it is not to be a subargument in a DB preprocessor function reference. The user may not declare any attributes for the ULIST function; the follcwing statement will be generated automatically:

DECLARE ULIST ENTRY(POINTER) RETURNS(POINTER);

Argument:

The list-pointer argument must be a pointer expression that identifies a list of keys in main storage. It must have been set when the ULIST function is invoked.

Result:

The value returned by the ULIST function is a pointer identifying the new list having only unique key values. However, if the argument list is found to not have any duplicated key values, its list pointer is simply returned (this always happens when the argument list is null or has only one key).

'The UNLOCK Statement'

The UNLOCK statement makes a locked current record or subrecord available to other tasks for READ operations; it may cause transmission of the current record or subrecord to the data base.

General Format:

UNLOCK FILE (file-name) <SUBFILE(ctlfield)>;

Syntax Rule:

The UNLOCK statement must be a subargument in a DB preprocessor function reference.

General Rules:

1. The FILE clause specifies the data base file whose current record is to be unlocked. The file must have the UPDATE attribute.

2. A record can be unlocked only by the task which locked it.

3a. If no SUBFILE clause is present, the current root record will be unlocked.

3b. A SUBFILE clause, if present, specifies that the current subrecord of a subfile is to be unlocked. The ctlfield expression value is converted to a character string the first eight characters of which identify the control field.

4. If the locked current (sub)record has been updated by a PUT or REPUT FIELD statement, the UNLOCK statement will cause it to be retransmitted to the data base. It continues to be the current (sub)record of the file, but PUT and REPUT statements are invalid until another current (sub)record is established.

5. Unlocking a (sub)record that was READ with the NOLOCK option or that has already been UNLOCKed has no effect.

'The UPLIST Function'

Unique Parent LIST builds a list of the unique parent (root) record keys from a given sub-record (children) key list and returns a pointer value identifying the new list to the point of invocation. The new list has the same number of parent keys as the number of subrecord keys in the given list. Parent keys will not be repeated, even if more than one of the given subrecord keys are dependent on a particular parent record. A previously current and updated subrecord of the subfile referenced by the given list will be transmitted to the data base. The subrecord identified by the last key in the given list will remain as the current subrecord of that sub-file; any current root or index records or subrecords of other subfiles will remain current. The READ cursor of the given list will be reset.

Reference:

```
UPLIST (file-name, child-list-pointer);
```

An UPLIST function reference is used as or in an expression; it is not to be a subargument in a DB preprocessor function reference. The user may not declare any attributes for the UPLIST function; the following statement will be generated automatically:

```
DECLARE UPLIST ENTRY(,PTR) RETURNS (PTR);
```

Arguments:

The file-name argument specifies the data base file from which subrecords are to be transmitted. It may not be an OUTPUT file. If the file is not open, it will be opened automatically. The file-name must be used in at least one DBEL/I statement elsewhere in the program.

The child-list-pointer argument must be a pointer expression that identifies a list in main storage of subrecord keys from the data base accessed by file-name. It must have been set when the UPLIST function is invoked.

Result:

The value returned by the UPLIST function is a pointer identifying the new unique parent list. The new list will be in order of ascending internal parent key values without duplicated values. If the given subrecord list is null, a NULL pointer value will be returned.

'The WRITE Statement'

The WRITE statement causes a physical record (presumably, from a backup file) to be transmitted to a SEQUENTIAL OUTPUT file.

General Format:

```
WRITE FILE (file-name) FROM (variable);
```

Syntax Rule:

The WRITE statement must be a subargument in a DB preprocessor function reference.

General Rules:

1. The FILE clause specifies the file to which the record is to be transmitted. It must be a SEQUENTIAL OUTPUT file owned by the user who executes the WRITE statement.

2. If the file is not open, it is opened automatically with the SEQUENTIAL OUTPUT attributes.

3. The variable in the FROM clause, declared and filled by the user, contains the record to be written. It must have the self-defining format of an internal variable-length record. Its key field value (without validation or conversion) must be higher, in order of ascending internal values, than that of the record transmitted by the previous WRITE statement for the file. (The record does not become the current record of the file for purposes of PUT statements.)

'The #XREF Function'

#XREF calculates the number of cross reference keys in the current record of an index and returns it to the point of invocation.

Reference:

```
#XREF (file-name, indfield)
```

A #XREF function reference is used as or in an expression; it is not to be a subargument in a DB preprocessor function reference. The user may not declare any attributes for the #XREF function; the following statement will be generated automatically:

```
DECLARE #XREF ENTRY(,CHAR(8)) RETURNS (FIXED BIN(31));
```

Arguments:

The file-name identifies a data base file. It may not be an OUTPUT file.

The indfield specifies the index file. A current index record must have been established by a READ INDEX statement when the #XREF function is invoked. The indfield expression value is converted to a character string, if necessary, the first eight characters of which identify the indexed field.

Result:

The value returned by the #XREF function is a binary integer of maximum precision giving the number of cross-references in a current index record. If an index record is not current, a zero value will be returned.

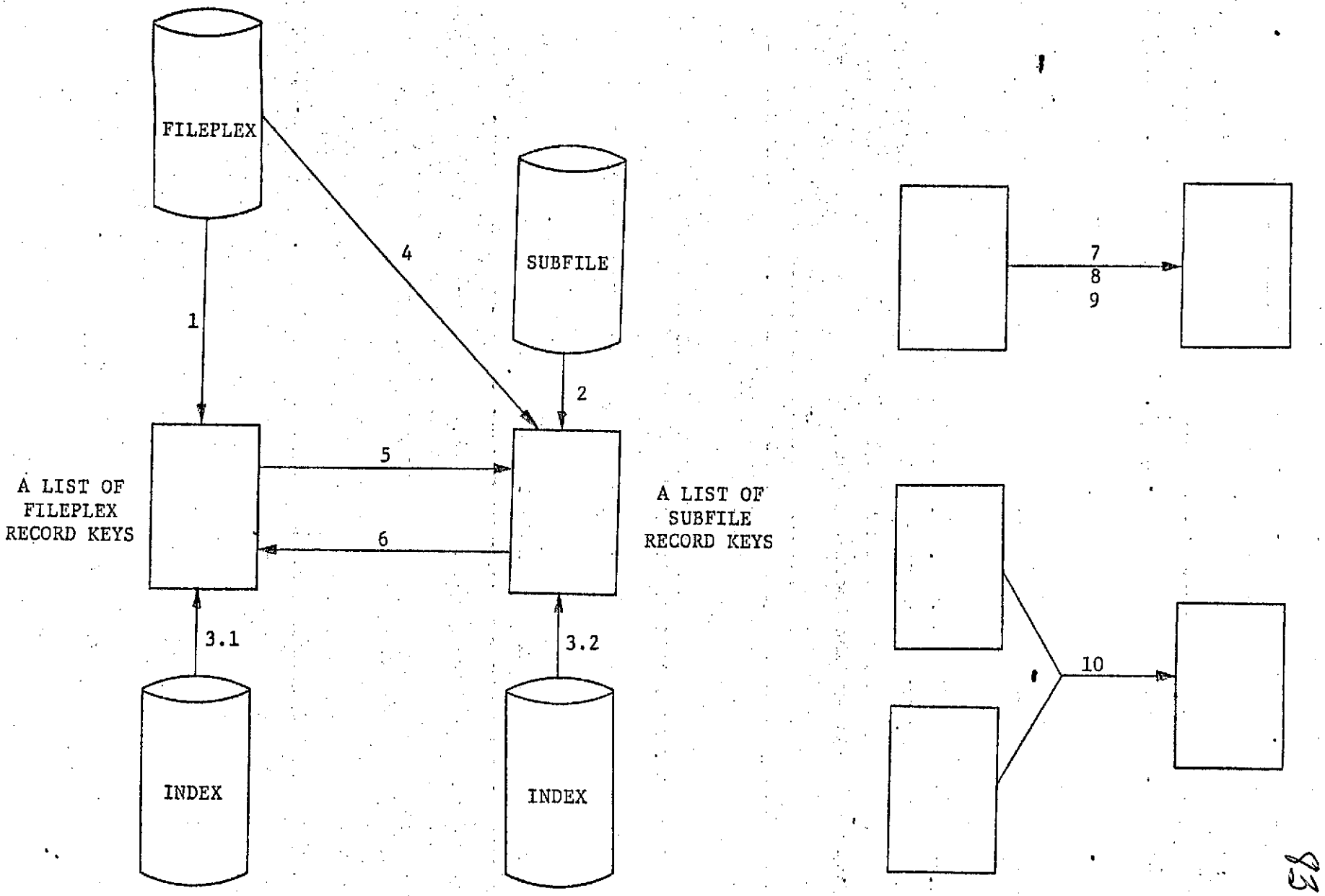


FIGURE 1. FORMATION OF LISTS

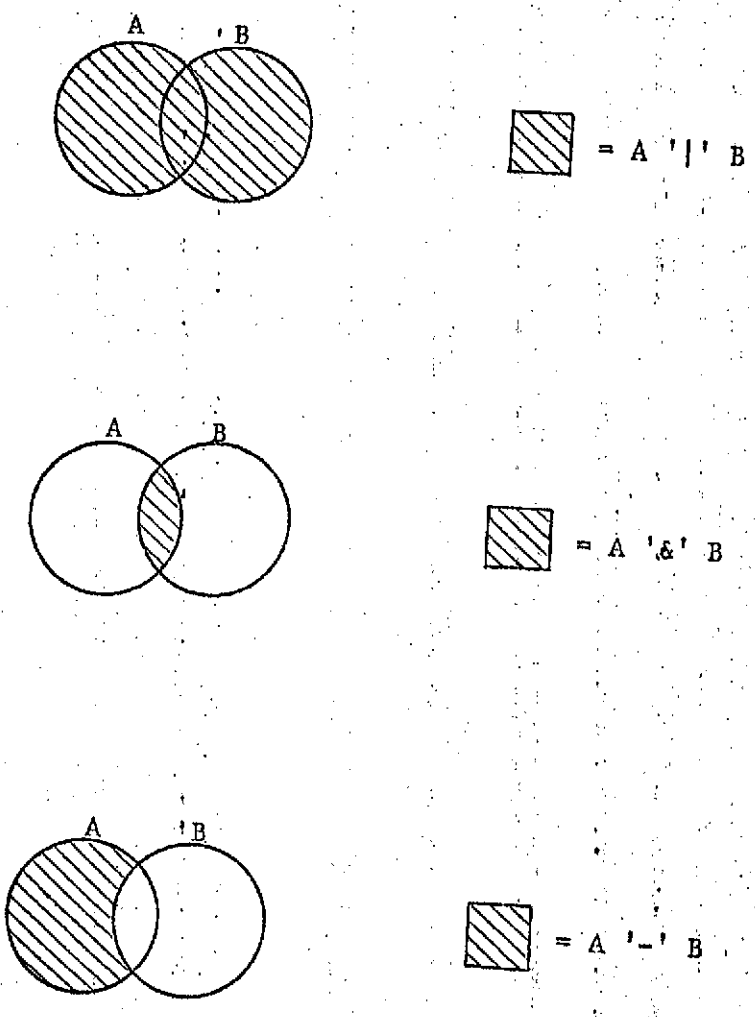


FIGURE 2. VENN DIAGRAMS

APPENDIX A.

FILE LEVEL STATEMENTS

```
ON ERRORFILE (mfcb) |_SYSTEM      _|
                    |_GO TO label_| ;

OPEN FILE (mfcb) <TITLE ('mfct')> |_DIRECT     _| |_INPUT  _|
                                  |_SEQUENTIAL_| | OUTPUT |;
                                                 |_UPDATE_|

CLOSE FILE (mfcb) <ERASE>;
```

RECORD LEVEL STATEMENTS

```
LOCATE FILE (mfcb) |_KEYFROM (expr)_|
                   |_SUBFILE (scfn)_| ;

READ FILE (mfcb) |_forwards             _|  <NOLOCK>;
                 | KEY (expr)             |
                 | LIST (ptr)  <KEY(n)> |
                 |_PER SUBFILE (scfn)  _|

READ FILE (mfcb) SUBFILE (scfn)  |_forwards  _|  <NOLOCK>;
                                 |_KEY (expr)_|

READ FILE (mfcb) INDEX (ifn)|_forwards  _| ;
                            |_KEY (expr)_|

UNLOCK FILE (mfcb) <SUBFILE (scfn)>;
```

PHYSICAL RECORD STATEMENTS

```
GET FILE (mfcb) RECORD INTO (var);

WRITE FILE (mfcb) FROM (var);
```

FIELD LEVEL STATEMENTS

```
PUT FILE(mfcb) FIELD(fn<,fn2>) FROM (expr<,expr2>);

GET FILE(mfcb) FIELD(fn<,fn2>) INTO (var<,var2>);

GET FILE(mfcb) INDEX(ifn) KEY INTO(var);

REPUT FILE(mfcb) FIELD (fn<,fn2>) FROM (expr<,expr2>);

fullword = #FIELD (mfcb, fn);

fullword = #XREF (mfcb,ifn);
```

DATABASE LIST STATEMENTS

```
GET FILE (mfcb)  ¦ SUBFILE (scfn)    ¦ LIST SET (ptr);
                 ¦ INDEX (ifn)       ¦
                 ¦_anchor is index _¦

GET FILE (mfcb)  <SUBFILE (scfn)> KEY SET (ptr);

Ptr = CCLIST (mfcb, scfn, ptr1);

Ptr = CPLIST (mfcb, ptr1);

Ptr = UPLIST (mfcb, ptr1);
```

NON-DATABASE LIST STATEMENTS

```
ON LISTERROR  ¦ SYSTEM      ¦
```

```
|_GO TO label_| ;

GET LIST (ptr) KEY (0);

GET LIST (ptr) KEY <(n)> INTO (var);

GET LIST (ptr1) KEY SET (ptr2);

Ptr = ULIST (ptr1);

Ptr = DUPLIST (ptr1);

Ptr = LIST (ptr1,op,ptr2);

SET LIST (ptr2) SIZE (dim) LIKE LIST (ptr1);

GET LIST (ptr1) INTERNAL KEY INTO (var);

PUT LIST (ptr2) INTERNAL KEY FROM (expr);

Fullword = #LIST (ptr);

FREE LIST (ptr <,ptr2>);

FREE LIST;
```

GLOSSARY

dim	an expression resulting in a numerical dimension value
expr	an expression resulting in a value
fn	an expression resulting in a field name
ifn	an expression resulting in an indexed field name
mfcb	mainline FILE control block name
n	an expression resulting in a numerical subscript value
op	list operator: '|' or '&' or '-'
ptr	pointer to a list of keys in main stroage
scfn	an expression resulting in a subfile control field name
var	variable data area name

TOPIC C.4 - DBJOIN - JOINING NEW USERS

I. INTRODUCTION

The JOIN command gives the NASIS Data Base Administrator the ability to control the access of retrieval users to the various files of the system. In addition, it also allows the DBA to specify passwords, time slice values and authority codes which influence use of the system. The information is maintained in data set NASIS.USERIDS.

II. COMMANDS:

JOIN

The JOIN command establishes a new NASIS-ID which can be used to access the system. This is accomplished by creating a new record in the data set and inserting the values for the various data fields.

Command: JOIN
Operands: NASISID=id,PASSWORD=code,TS=value,
AUTH=authority,FILES=file list

Where:

id
identifies the new NASIS-ID to be created.

Specified as: a 1-8 character alphanumeric value beginning with a letter.

code
identifies the password or indentification code to be used for this NASIS-ID.

Specified as: a 1-8 character alphanumeric value.

Default: No password will be assigned.

value
indicates the magnitude of the time slice in Milli Seconds to be assigned to this NASIS-ID under MTT mode of operation.

Specified as: a 1-5 digit numeric value.

authority
indicates the authority level to be assigned to this NASIS-ID under MTT mode of operation.

Specified as: a one character code, 'U' for user or 'D' for DBA.

Default: 'U' will be assigned.

file list
identifies the files to be made available to this NASIS-ID.

Specified as: a list of fully qualified file names, i.e. DBA-ID.FILE-ID.

QUIT:

The QUIT command removes a NASIS-ID from the list of valid ids.

Command: QUIT
Operand: NASISID=id

CHANGE:

The CHANGE command is used to alter the values of one or more of the data fields (other than the file list) associated with a particular NASIS-ID.

Command: CHANGE
Operands: NASISID=id,PASSWORD=code,TS=value, AUTH=authority

ADD:

The ADD command is used to specify new files which are to be added to the list of files to which a given NASIS-ID is permitted access.

Command: ADD
Operands: NASISID=id,FILES=file list

DELETE:

The DELETE command is used to remove files from the list of files to which a particular NASIS-ID is permitted access.

Command: DELETE
Operands: NASISID=id,FILES=file list

DISPLAY:

The DISPLAY command is used to list the files available to a particular NASIS-ID, along with the other data values present in his identification record.

Command: DISPLAY
Operand: NASISID=id

III. EXAMPLE

USER: join john,ace,99999,,
SYSTEM: JOHN JOINED TO NASIS WITH PASSWORD=ACE, TIMESLICE=99999 MILLESECONDS, AND AUTHORITY=.
USER: add john,(safety.asrdi,safety.erts)
SYSTEM: Adds the two files to the list of files available to JOHN.
USER: display john
SYSTEM: Display the current information maintanined for JOHN.
USER: change john,auth=d
SYSTEM: Applies the appropriate change.

TOPIC D.2 - DESCRIPTOR EDITOR

I. INTRODUCTION

The Descriptor Editor is an editing program used for creating and updating the field descriptors of a NASIS Data Base.

II. INVOKING THE EDITOR

The Descriptor Editor is invoked by entering the EDIT command and specifying the appropriate mode of operation and the descriptor file to be edited.

EDIT MODE=<CREATE|UPDATE|RESTORE>,FILE=filename

Where:

MODE
Is Specified as:

CREATE: assumes that no data files exist and that the user is either creating or continuing to create field descriptors.

UPDATE: assumes that data files do exist and that the user wishes to modify the description of one or more of the fields. The UPDATE mode allows the user to make changes that do not affect the physical format of the record.

RESTORE: reads in previously, check-pointed descriptors and continues processing in the CREATE mode.

FILE
Is specified as the name of the data base descriptors the user wishes to edit. Specified as an alphanumeric string of at most 6 characters, the first of which must be alphabetic.

For all modes the first letter of the mode type is a sufficient abbreviation. If the entered mode value is invalid, the editor will re-prompt the user for a correct value. If the user defaults the prompt for the mode, the Editor will terminate and control will be returned to the MTT director.

EXAMPLES:

1. The user wants to create a new data base whose name is RECPIE.

```
SYSTEM:  ENTER NASIS COMMAND:
SYSTEM:  ENTER:
USER:    EDIT
SYSTEM:  ENTER MODE:
USER:    CREATE
SYSTEM:  ENTER FILE NAME:
USER:    PEOPLE
```

2. The user wants to modify the descriptors for an existing data base whose name is PGMS.

```
SYSTEM:  ENTER NASIS COMMAND:
SYSTEM:  ENTER:
USER:    EDIT UPDATE,PGMS
```

3. The user has a checkpointed set of descriptors for the data base GAMES which he wishes to continue defining.

```
SYSTEM:  ENTER NASIS COMMAND:
SYSTEM:  ENTER:
USER:    EDIT RESTORE,GAMES
```

III. DEFINITIONS

The following definitions are used throughout this section:

1. Boolean Values - Used where ever a yes or no type of response is required. The following are acceptable values for a 'yes' type of response:

   YES, Y, TRUE, T, ON, 1.

   The following are acceptable values for a 'no' type of response:

   NO, N, FALSE, F, OFF, 0.

2. Fieldname - Is a character string of 1-8 characters long of the following form: the first character must be alphabetic, and the other characters, if any, must be alphanumeric.

3. Routine Name - Is a character string of 1-8 characters long with the following form: the first character must be alphabetic, and the rest of the characters, if any, must be alphanumeric.

IV. THE CREATE MODE COMMANDS

A. The ADD and CHANGE COMMANDS allow the user to create a new field descriptor or modify existing

field descriptors.

```
ADD(|CHANGE) FLDNAME=field-name,
          TYPE=(FLDTYPE=field-type
                <,ALIGN=<RIGHT|LEFT>>),
          FORM=(FLDFORM=field-format,
                FLDLEN=field-length,
                ELEMLEN=element-length,
                ELEMLIM=element-number
                <,UNIQUE=<Y|N>>),
          ROUTINES=(CONV=conversion-routine,
                    FORMAT=formatting-routine,
                    VALID=validation-routine,
                    VALIDARG=validation-argument),
          INDEXED=(INDEX=<Y|N>,
                   IFLDNAME=field-name
                   <,EXTINT=<INTERNAL|EXTERNAL>,
                     EXTLEN=external-length,
                     SPANNED=<Y|N>>),
          ASSOCED=(ASSOC=<Y|N>,
                   AFLDNAME=field-name),
          SUBFILED=(SUBFILE=<Y|N>,
                    SFLDNAME=field-name),
          SUBFIELD=(SUBFLD=<Y|N>,BASEFLD=field-name,
                    OFFSET=offset
                    <,<FILE=<*filename|ANCHOR>>
                    or <FILE=<ASSOCIATED|SUBFILE>,
                        FLDNAME2=field-name>>)
```

Where:

FLDNAME
identifies the field to be added.

Specified as: a valid fieldname.

FLDTYPE (FIELD TYPE)
identifies the physical format of the field.

Specified as:

A - alphanumeric character string

B - bit string

BN - 8 bit unsigned binary number

BP - packed bit string. These fields will be placed immediately after the key field as one contiguous bit string.

HX - hexadecimal

LN - large numberic (32 bit signed binary number).

S - scientific (14 digit decimal number within the range of 10**-75 : 10**+75).

SD - scaled decimal (nine digit numbers within the range 10**-9 : 10**+9).

SN - short numeric (16 bit signed binary number).

SS - short scientific (six digit decimal number within the absolute range of 10**-75 : 10**+75).

ALIGN (ALIGNMENT)
identifies right or left alignment of the field.

Specified as: 'RIGHT' or 'R' for right alignment and 'LEFT' or 'L' for left alignment.

FLDFORM - (FIELD FORMAT)
identifies the logical format of the field.

Specified as: F-FIXED, V-VARIABLE, FE-FIXED ELEMENT, VE-VARIABLE ELEMENT.

FLDLEN (FIELD LENGTH)
indicates the length of fixed fields or the maximum length for other types of fields.

Specified as: a positive number.

(1) For the file key field, the maximum field length is 256.

(2) For all other fields:

(a) If FLDFORM=F, then the maximum field length is 3996 minus the key field length;

(b) For all other values of FLDFORM, the maximum length is 3994 minus the key field length.

ELEMLEN (ELEMENT LENGTH)
indicates the maximum length of fixed and variable elements.

Specified as: a positive number with the range of 1-256 if FLDFORM is FE: the range is 1-255 if FLDFORM is VE.

ELEMLIM
indicates the maximum number of elements allowed in the field.

Specified as: a positive number.

(1) If FLDFORM=FE, then the maximum number of elements is equal to the field length.

(2) If FLDFORM=VE, then the maximum number of elements is the field length divided by two.

UNIQUE
indicates whether or not all element values within a multi-element field are to be unique.

Specified as: a boolean value.

Default: N

CONV (CONVERSION ROUTINE NAME)
identifies the name of the routine used to convert the input data as it is placed into the data base.

Specified as: a routine name.

FORMAT (FORMATTING RCUTINE NAME)
identifies the routine used to format the data for output from the data base.

Specified as: a routine name.

VALID (VALIDATION RCUTINE NAME)
identifies the name of the routine used to validate the input data.

Specified as: a routine name.

VALIDARG (VALIDATION ROUTINE ARGUMENT)
indicates the argument required by the validation routine to validate the input

values.

Specified as: a hexadecimal character string of 1-100 characters.

INDEX
indicates whether the field is to be indexed.

Specified as: a boolean value.

Default: N

IFLDNAME
identifies anctber field previously defined with which this field is to be indexed.

Specified as: a valid fieldname of a previously entered indexed field.

Default: the Editor assumes that this field is the first entered field of a new index file.

EXTINT
indicates whether the key of the index file is to be in internal or external form. If the key values are to be in external form, then the field values must be formatted before being placed on the index file.

Specified as: INTERNAL or I for internal form or EXTERNAL or E for external form.

Default: internal form, i.e., the value used on the index file is the same as that stored in the anchor file.

EXTLEN (EXTERNAL LENGTH)
indicates the maximum lenqth possible for an formatted value of the external field.

Specified as: a positive numeric value in the range 1-256.

NOTE: if the EXTINT entered value is external, then EXTLEN must be specified.

SPANNED
indicates that this index is to consist of spanned records.

Specified as: a boolean value.

Default: N

NOTE: this implies that the maximum length for index keys can be no larger than 255 to allow for a one byte spanned counter.

ASSOC (ASSOCIATED)
indicates whether the field is to be associated.

Specified as: a boolean value.

Default: N

AFLDNAME
identifies another field previously defined with which this field is to be associated.

Specified as: a valid previously entered field name.

Default: the Editor assumes that this field is the first entered field of a new associated file.

SUBFILE
indicates whether the field is to appear on a subfile.

Specified as: a boolean value.

Default: N

SFLDNAME
identifies another field previously defined which identifies the subfile on which the field is to be placed. The field named may be the subfile control field.

Specified as: a valid previously defined fieldname.

SUBFLD
Indicates whether this field is to be defined on either a part or the whole of another field.

BASEFLD
identifies the field on which this new field is to be defined.

Specified as: a valid previously defined fieldname.

OFFSET
indicates the bit or character position of the defined field on which this subfield is to start.

Specified as: a positive numeric value between zero and the lenght of the defined field minus one.

NOTE: the offset value must be specified if the subfield is specified.

FILE
identifies the descriptor region on which resides the field that is the defining base for this subfield.

Specified as:

(1) The character '*' concatenated to the descriptor file region name.

(2) The anchor file which may be entered as either of the following: ANCHOR or AN.

(3) An associated file which may be entered as either of the following: ASSOCIATED or AS.

(4) A subfile which may be entered as either of the following: SUBFILE or S.

Default: will be assumed to be the anchor file.

NOTE: this parameter only needs to be entered if the defined fieldname is not unique within the data base, such as RECLEN.

FLDNAME2
identifies a field which is used to determine which associated file or which subfile is being referenced.

Specified as: a valid fieldname.

NOTE: Any parameter to the CHANGE function which is defaulted, will imply that the existing value for that descriptor field will be left unaltered.

NOTE: There is a user default variable "EDPROMPT" which when set equal to "Y" will cause the user to be prompted for every possible applicable parameter while the user is either ADDing a new field or CHANGing an existing field. In the normal mode there are parameters such as field alignment ("ALIGN") which are not prompted for if the user does not enter them in the command stream.

EXAMPLES:

1. When first creating a new set of descriptors, the user is first prompted for the anchor file key field.

```
SYSTEM:  ENTER KEY:
USER:    ADD ACCESSNO
SYSTEM:  ENTER FIELDTYPE:
USER:    A
SYSTEM:  ENTER FIELD FORMAT:
USER:    F
SYSTEM:  ENTER FIELD LENGTH:
USER:    8
SYSTEM:  ENTER ROUTINES:
USER:    (return - wants standard defaults)
SYSTEM:  ENTER:   (prompt for next editing
                   request)
```

NOTE: If the user declines to enter the key field information, the Editor is terminated and control is returned to the Maintenance director.

2. The user wishes to add the field USERNAME which is to be a varying element field, each element is to be 12 characters long and allow for 50 elements per record. USERNAME is to be placed on the associated file along with USERTYPE. It is also to be inverted.

```
SYSTEM:  ENTER:
USER:    ADD
SYSTEM:  ENTER FIELD NAME:
USER:    USERNAME
SYSTEM:  ENTER FIELD TYPE:
USER:    A
SYSTEM:  ENTER FIELD FORMAT:
USER:    VE
SYSTEM:  ENTER FIELD LENGTH:
USER:    500
SYSTEM:  ENTER ELEMENT LENGTH:
USER:    12
```

```
SYSTEM:  ENTER NUMBER OF ELEMENTS:
USER:    50
SYSTEM:  ENTER ROUTINES:
USER:    (CONV=UNCVT,FORMAT=UNFMT,
          VALID=UNVAL,)
SYSTEM:  IS FIELD TO BE INDEXED?
USER:    YES
SYSTEM:  ON WHICH INDEX FILE IS FIELD TO BE
         PLACED?
USER:    (return)
SYSTEM:  IS FIELD TO BE ON AN ASSOCIATED
         FILE?
USER:    Y
SYSTEM:  ON WHICH ASSOCIATED FILE IS FIELD TO
         BE PLACED?
USER:    USERTYPE
SYSTEM:  IS FIELD TO BE PLACED ON A SUBFILE?
USER:    NO
SYSTEM:  ENTER DEFINING BASE FIELD NAME:
USER:    (return)
SYSTEM:  ENTER:
```

3. The user wishes to change the field length on field SOCSECNC from 8 to 9 and wishes to make the index on which it appears a spanned index.

```
SYSTEM:  ENTER:
USER:    CHANGE SOCSECNC,,(,9),,(,,,,Y),,,,
```

E. The ADDLIKE Descriptor Function

This function allows the user to create a descriptor with all the same specifications as a previously defined field.

```
ADDLIKE FLDNAME1=new-fieldname,
        FLDNAME2=other-fieldname
```

Where:

FLDNAME1
identifies the new descriptor to be added.

Specified as: a valid fieldname.

FLDNAME2
identifies a previously defined field of which the new field is to be an exact duplicate except for the field name.

Specified as: a valid field name.

EXAMPLE:

1. The user wishes to add field MINKEYWD to have exactly the same specifications as the field MAJKEYWD.

```
SYSTEM:  ENTER:
USER:    ADDLIKE MINKEYWD,MAJKEYWD
```

C. The CHECKPOINT Command

Checkpoint allows the user to save the descriptors currently defined in a separate TSS VAM file.

```
CHKPOINT  (none)
```

CHKPOINT should be used when it is deemed necessary to save the descriptors as rapidly as possible. The user may continue to process at a future time VIA the Restore Command.

D. The CREATESUB Command

The command allows the user to create a subfile.

```
CREATSUB FLDNAME=control-field-name,
         MAXRECS=#-records,
         ASSOC=<Y|N>,
         AFLDNAME=field-name
```

Where:

FLDNAME
identifies the field to be known as the subfile control field.

Specified as: a valid field name.

MAXRECS
indicates the maximum number of subfile records that can occur per anchor file record.

Specified as: a binary number in the range of 1:1325.

ASSOC
indicates whether the field is to be associated.

Specified as: a boolean value.

Default: N

AFLDNAME
identifies another field, previously defined, with which this field is to be associated.

Specified as: a valid previously entered fieldname.

EXAMPLE:

The user wants to create a subfile for "PETS" which is to be associated with CHILD.

```
SYSTEM:  ENTER:
USER:    CREATSUB PETS,20,Y,CHILD
```

E. The DELETE Command

This command allows the user to delete a previously created field descriptor other than the key field.

DELETE FLDNAME=fieldname

Where:

FLDNAME
identifies the field to be deleted.

Specified as: previously described field name.

F. THE DISPLAY COMMAND

This command allows the user to display the specifications entered for a previously created descriptor.

DISPLAY FLDNAME=fieldname

Where:

FLDNAME
identifies the field descriptor to be displayed.

Specified as: a valid fieldname.

G. The END command

This command terminates a descriptor editor session.

```
END    (none)
```

After the END command has finished, control will be returned to the Maintenance director. If the user has not FILE'd since making additions, deletions, or modifications, he will be queried as to whether he wishes to FILE the descriptors. If the user wishes to terminate, then the descriptor editor will indeed terminate the current session; otherwise, the user will be prompted for his next descriptor editor command.

H. The FIELDS Command

This command allows the user to display the names of all the field descriptors thus far defined.

```
FIELDS    (none)
```

I. The FILE Command

This function allows the user to indicate that he wants the descriptors to be written from virtual memory to disk storage.

```
FILE DESCOK=<Y|N>
```

Where:

DESCOK
: indicates whether or not the descriptors are complete. If a NO value is indicated no data can be loaded into this file.

  Specified as: a boolean value.

  Default: N

J. The FLDSEC (Field Security) Command

This command permits the data base owner to restrict access to a field or a group of fields.

```
FLDSEC FLDNAME=(field-name),
       SECURITY=(<<ADD|DELETE>.>
                 security-code<,...>)
```

Where:

FLDNAME
: is a list of one or more existing fieldnames to which the data base owner wishes to restrict access.

Specified as: a list of valid fieldnames.

SECURITY
is a list of security codes appended by an add-delete code separated from the security code by a period. The add-delete code is specified as A or ADD for adding a security code and D or DELETE for deleting a security code. If no add-delete code is entered, it is assumed the user is adding the security code. The security code is specified as an alphanumeric character string of 1 to 8 characters. A maximum of 18 security codes may be specified for any field.

EXAMPLE:

The data base owner wishes to restrict the fields ACCOUNT and VALUE to the persons with the security codes BOB, HARRY, and JOHN and to delete TOM from the security list.

```
SYSTEM: ENTER:
USER:   FLDSEC (ACCOUNT,VALUE),(ADD.BOB,
                A.HARRY,A.JOHN,D.TOM)
```

K. The MOVE Command

This command allows the user to reposition fields within the defined data layout.

```
MOVE FLDNAME1=new-location-fieldname,
     FLDNAME2=fieldname
```

Where:

FLDNAME1
identifies which field or the new location after which the field specified by FLDNAME2 is to be positioned.

Specified as: a valid fieldname.

FLDNAME2
identifies the field to be moved.

Specified as: a valid fieldname.

NOTE: A redefined field, i.e., subfield, cannot be moved as its position is determined by the position of the base field. If a subfield is specified as the new position fieldname, the MOVE command will locate and

use the base field name as the new position field name.

NOTE: A superfield cannot be used as a new position fieldname, nor can it be moved, as a superfield consisting only of other fields has no field position.

EXAMPLE:

The user has entered the three fixed fields in the following: AREACODE, LOCALNUM, EXCHNG The user wishes to change the order to AREACODE, EXCHNG, LOCALNUM.

```
SYSTEM: ENTER:
USER:   MOVE AREACODE,EXCHNG
```

Notice this could also be accomplished by the following:

```
SYSTEM: ENTER:
USER:   MOVE EXCHNG,LOCALNUM
```

L. The PRINT Command

This command generates a printer listing of all the field descriptors and file descriptors as they exist in core at the time the PRINT was issued.

```
PRINT   (none)
```

M. The RENAME Command

This command permits the user to change the name of a field without altering any of its other specifications or its location in the data record.

```
RENAME FLDNAME1=new-fieldname,
       FLDNAME2=old-fieldname
```

Where:

FLDNAME1
identifies the new field name.

Specified as: a valid fieldname.

FLDNAME2
identifies the existing field name

Specified as: a valid fieldname.

EXAMPLE: The user wishes to change the name of the field OLDNAME to the name NEWNAME.

```
SYSTEM: ENTER:
USER:   RENAME NEWNAME,OLDNAME
```

N. The RECSEC (Record Security) Command

This command permits the user to control access to a group or groups of records within the data base.

```
RECSEC DFLDNAME=field-name,
       SECURITY=(<<ADD|DELETE>.>
                 security-code:mask<,...>)
```

Where:

DFLDNAME
is the existing fieldname to which the file record security is to apply.

Specified as: a valid fieldname.

SECURITY
is a list of up to 18 security codes and security masks determining who is to be permitted access to the secured records on the file. It is specified as an add-delete code followed by a period, followed by the security code, followed by a colon, followed by the security mask. The add-delete code is specified as ADD or A for adding a security code, or DELETE or D for deleting a security code. The security code is an alphanumeric character string of 1-8 characters. The mask is two digit hexadecimal code.

The security code is used to compare against the value in the record security field of a record to determine whether or not a user has access to that record.

O. The RESTORE Command

This command permits the user to restore to core memory the descriptors which had been previously saved by the use of the CHKPCINT command.

```
RESTORE   (none)
```

P. The SAVSTRT (Save Strategy) Command

This ccmmand allows saving of descriptor editor commands in the strategy data set for future recreation of descriptors as they existed in virtual memory when the SAVSTRAT command was issued.

```
SAVSTRT STRTNAME=strategy-name
```

Where:

STRTNAME
is the strategy name in the strategy data set in which the descriptor editor commands are to be saved.

Specified as: a 1-16 character long alphanumeric character string.

O. The Superfld (Define Superfield) Command

This command allows the user to create a new field descriptor which is defined as consisting of Data from several fields.

```
SUPERFLD FLDNAME=fieldname,
         ROUTINES=FORMAT=formatting-routine,
         FLDLIST=(<<INTERNAL|EXTERNAL>,>
                  field-name<,...>)
```

Where:

FLDNAME
identifies the name of the new superfield.

Specified as: a valid field name.

FORMAT
identifies the routine used to format the data for output from the data base.

Specified as: a routine name.

FLDLIST
is a list of the previously defined fieldnames from which this superfield is to be composed. The order of the fieldnames used to define the superfield is the order in which they were entered. The user may specify whether the internal or external form of the field is tc be passed to the superfield formatting routine.

Specified as: a list of up to 16 character

strings of the form: The output format concatenated to a period concatenated to the fieldname to be included in the superfield. The format type internal may be specified as:

INTERNAL or I

The format type external may be specified as:

EXTERNAL or E

Default: If the output format is omitted, then it will be assumed to be the external format type.

NOTE: The superfield components must stay within the following restrictions:

1. It may contain at most one multi-element field.

2. It may contain components from one but not more than one subfile.

IV. THE UPDATE MODE COMMANDS

A. The CHANGE COMMAND

This command allows the user to modify a previously defined field.

```
CHANGE FLDNAME=fieldname,
       TYPE=(FLDTYPE=field-type
             <,ALIGN=<RIGHT|LEFT>>),
       FORM=(FLDFORM=field-format,
             FLDLEN=field-length,
             ELEMLEN=element-length,
             ELEMLIM=element-number
             <,UNIQUE=<Y|N>>),
       RCUTINE=(CONV=conversion-routine,
                FORMAT=formatting-routine,
                VALID=validation-routine,
                VALIDARG=validation-argument)
```

Where:

FLDNAME
identifies the field to be modified.

Specified as: a valid fieldname.

FLDTYPE
identifies the physical format of the field.

Specified as:

A for an alphanumeric character string, of which each character may consist of any valid EBCDIC character.

B for a bit string.

BN for an 8 bit unsigned binary number which has a value in the range 0-255.

BP for a packed bit string the same as B, except that these fields will be placed immediately after the key field as one continuous bit string.

HX for a string of hexadecimal numbers.

LN for numeric or a 32 bit signed binary number.

S for scientific or 14 digit decimal number within the range of 10**-75 : 10**+75.

SD for scaled decimal nine digit number within the range of 10**-9 : 10**+9.

SN for numeric or 16 bit signed binary number.

SS for short scientific or a six digit decimal number within the range of 10**-75 : 10**+75.

ALIGN
identifies either right or left alignment of the field.

Specified as: RIGHT or R for right alignment and LEFT or L for left alignment.

FLDFORM
identifies the logical format of the field.

Specified as: F for FIXED,V for VARIABLE,FE, for FIXED ELEMENT,VE, for VARIABLE ELEMENT.

FLDLEN

indicates the length of fixed fields or the maximum length for other types of fields.

Specified as: a positive integer.

(1) For the anchor file key field, the maximum field length is 256.

(2) For all other fields:

(a) If FLDFORM=F, then the maximum field length is 3996 minus the key field length; otherwise,

(b) For all other values of FLDFORM, the maximum length is 3994 minus the key field length. This allows for a two byte field length indicator.

ELEMLEN
indicates the length of fixed elements or the maximum length for variable elements.

Specified as: a positive numeric value with the range of 1-256 if FLDFORM is FE, else the range is 1-255 if FLDFORM is VE. This allows one byte for an element length indicator.

ELEMLIM
indicates the maximum number of elements allowed in the field.

Specified as: a positive integer.

(1) If FLDFORM=FE, then the maximum number of elements is equal to the field length.

(2) If FLDFORM=VE, then the maximum number of elements is the field length divided by two.

UNIQUE
indicates whether or not all element values within a multi-element are to be unique.

Specified as: a boolean value.

CONV
identifies the name of the routine used to convert the input data as it is placed into the data base.

Specified as: a routine name.

FORMAT
identifies the routine used to format the data for output from the data base.

Specified as: a routine name.

VALID
identifies the name of the routine used to validate the input data.

Specified as: a routine name.

VALIDARG
indicates the argument required by the validation routine to validate the input values.

Specified as: a hexadecimal character string of 1-100 characters.

NOTE: In the UPDATE mode, values to the CHANGE function will not be accepted which cause changes to be made to other field descriptor records, such as changing the field length if the field format is fixed as this changes the base length of the data records.

NOTE: Any parameter to the CHANGE function which is defaulted, will imply that the existing value for that descriptor field will be left unaltered.

Note: There is a user dafault veriable "EDPROMPT" which when set equal to "Y" will cause the user to be prompted for every possible applicable parameter while the user is CHANGE'ing an existing field. In the normal mode there are parameters such as field alignment ("ALIGN") which are not prompted for if the user does not enter them in the command stream.

EXAMPLE:

The user wishes to change the specifications for the field PEOPLE to RIGHT alignment, change the element length from 20 to 30 and the element limit from 5 to 10.

```
SYSTEM:  ENTER:
USER:    CHANGE PECPLE,(,RIGHT),(,,30,10),,
```

B. The DISPLAY COMMAND

This command allows the user to display the specifications entered for a previously created descriptor.

DISPLAY FLDNAME=fieldname

Where:

FLDNAME
identifies the field descriptor to be displayed.

Specified as: a valid fieldname.

C. The END COMMAND

The END command is terminates a descriptor editor session

END (none)

After the END command has finished, control will be returned to the Maintenance Director.

D. The FIELDS Command: displays all of the descriptor fieldnames in the descriptor file record, and all of the descriptor fieldnames in a field descriptor.

FIELDS (none)

E. FLDSEC (Field Security) Command: permits the file owner to restrict access to a field.

```
FLDSEC FLDNAME=field-name,
       SECURITY=(<<ADD|DELETE>.>
                 security-code<,...>)
```

FLDNAME
is an existing field name to which the owner wishes to restrict access.

Specified as: a valid fieldname.

SECURITY
is a list of security codes appended by an add-delete code separated from the security code by a period. The add-delete code is specified as A or ADD for adding a security code and D or DELETE for deleting a security code. If no add-delete code is entered, it

is assumed the user is adding the security code. The security code is specified as an alphanumeric character string of 1 to 8 characters. A maximum of 18 security codes may be specified for any field.

F. The PATCH Command

This command is used to change the value of almost any descriptor field on any descriptor record in any descriptor region. To use the PATCH command, the user must do a REVIEW of the desired descriptor record. This not only displays the contents of this descriptor but also positions to the record that is to be patched.

```
PATCH (keyword=text<,...>)
```

Where:

keyword
identifies the descriptor field that is to be patched.

text
is the value with which the descriptor field specified in 'keyword' is to be patched.

The user may specify any number of patches in a parenthesized list.

The following is a list of file descriptor or header descriptor field names that may be patched and their values.

	HEADER FIELDNAME	FIELD VALUES
(1)	FILETYPE	ANCHOR or 1, ASSOCIATE or 2, SUBFILE or 3, INDEX or 4.
(2)	DESCRCT	A positive integer <= 4000.
(3)	BSELNGTH	A positive integer <= 4000.
(4)	DESCOK	A boolean value.
(5)	SPANNED	A boolean value.
(6)	DATA	A boolean value.
(7)	MNTNABLE	A boolean value.
(8)	MNTNING	A boolean value.
(9)	LOADABLE	A boolean value.
(10)	RECSECFP	A positive integer <= 261.
(11)	RSECTYCD	The form of the patch text is: (n) security-code:mask

Where:

n

is the index of the security code to be patched. The index must be entered or the patch will be rejected.

Specified as: a positive integer <= 18.

NOTE: The next security code value may be added to the list by specifying the next larger index value.

Refer to the RECSEC command writeup for a discussion of the security parameter.

EXAMPLE:

The user wishes to patch the anchor header descriptor so that BSELNGTH=31, DATA=NO, and the second value of record security to BOB:60.

```
SYSTEM:  ENTER
USER:    REVIEW ' ',*HEADER
SYSTEM:  (displays the anchor header
          information.)
SYSTEM:  ENTER:
USER:    PATCH (BSELNGTH=31,DATA=N,
                RSECTYCD=(2)BOB:60)
SYSTEM:  ENTER
```

The following is a list of field descriptor fieldnames that may be patched along with their values.

	FIELD DESCRIPTOR FIELDNAMES	FIELD VALUES
(1)	ASSOCFIL	a one character string in the range '0' to '9'.
(2)	SUBFILE	a one character string in the range 'Q' to 'Z'.
(3)	INVFILE	a one character string in the range 'A' to 'P'.
(4)	READONLY	a boolean value.
(5)	SUBCNTRL	a boolean value.
(6)	VARFLD	VARYING or V,FIXED or F.
(7)	BITFLD	a boolean value.

(8) NUMALIGN — RIGHT or R,LEFT or L.

(9) VARELT — VARYING or V,FIXED or F.

(10) UNIQUELT — a boolean value.

(11) INDEXEXT — EXTERNAL or E,INTERNAL or I.

(12) GENERCRT — a routine name.

(13) VALIDRTN — a routine name.

(14) REFORMAT — a routine name.

(15) FLDPOSIT — a positive integer <= 4000.

(16) FLDLEN — a positive integer. If the field is indexed then the maximum value is 256. Otherwise the maximum value is 4000.

(17) ELTLIM — a positive integer. If the elements are fixed length, the maximum value is 4000. Otherwise the maximum value is 2000.

(18) ELTLEN — a positive integer <= 256.

(19) VALIDARG — a hexadecimal character string of length 1 to 100 characters.

(20) NAMEFLD — The patch text is of the form:

(n)<<INTERNAL|EXTERNAL>.>fieldname

Where:

n

is the index of the superfield component to be patched.

Specified as: a positive integer <= 16.

NOTE: The index must be entered or the patch will be rejected.

Refer to the SUPERFLD command writeup for the superfield components description.

(21) SECURITY — The patch is in the form:

(n) security-code

Where:

n
: is the index of the security code to be patched.

  Specified as: a positive integer <= 18.

  NOTE: The index must be entered or the patch will be rejected.

security-code
: is an alphanumeric character string of length 1 to 8 characters.

EXAMPLE:

The user wishes to patch the field PHONENUM on associate file 1 to have a formatting routine of PHONFMT on the third component of this superfield to be in external form and have the field name of LOCALNUM.

```
SYSTEM:  ENTER:
USER:    REVIEW 1,PHONENUM
SYSTEM:  (displays the field information.)
SYSTEM:  ENTER:
USER:    PATCH (REFORMAT=PHONFMT,
                NAMEFLD=(3)E.LOCALNUM)
SYSTEM:  ENTER:
```

G. The RECSEC (RECORD SECURITY) COMMAND

This command permits the owner to control access to a group or groups of records.

```
RECSEC DFLDNAME=field-name,
       SECURITY=(<<ALL|DELETE>,>
                 security-code<,...>)
```

Where:

DFLDNAME
is an existing fieldname which is used to define which file record security is to apply.

Specified as: a valid fieldname.

SECURITY
is a list of up to 18 security codes and security masks determining who is to be permitted access to the file. It is specified as an add-delete code, followed by a period, followed by the security code, followed by a colon, followed by the security mask. The add-delete code is specified as ADD or A for adding a security code, or DELETE or D for deleting a security code. The security code is an alphanumeric character string of 1-8 characters. The mask is a two digit hexadecimal code.

The security code is used to compare against the value in the record security field of a record to determine whether or not a user has access to that record.

NOTE: In the UPDATE mode the record security must already exist for the file to be able to use RECSEC. In the UPDATE mode, RECSEC is used to update the existing list of record security codes and masks.

H. The REVIEW COMMAND

This command is used to review the contents of any descriptor record on any descriptor file. This includes dummy records, file descriptor records and those records such as RECLEN which are not unique to the entire data base.

```
REVIEW FILE=file-name,
       FLDNAME=<*HEADER|field-name>
```

Where:

FILE
identifies the descriptor region containing the fieldname to be reviewed.

Specified as: the full descriptor region name or the character suffix of the

descriptor region.

NOTE: A null value is taken to indicate the anchor region.

FLDNAME
identifies the field which is to be reviewed.

Specified as: a valid fieldname or either of the following character strings: *HEADER or * which will imply a review of the file descriptor for the descriptor region named above.

APPENDIX A.

A. Descriptor Editor command format.

1. Edit Descriptor.

```
EDIT MODE = <CREATE|UPDATE|RESTORE>
```

B. Create Mode ccmmand formats.

1. ADD
```
FLDNAME=field-name,
     TYPE=(FLDTYPE=field-type
           <,ALIGN=<RIGHT|IEFT>),
     FORM=(FLDFORM=field-format,
           FLDLEN=field-length,
           ELEMLEN=element-length,
           ELEMLIM=element-number
           <,UNIQUE=<Y|N>>),
     ROUTINES=(CONV=conversion-routine,
               FORMAT=formatting-routine,
               VALID=validation-routine,
               VALIDARG=validation-argument),
     INDEXED=(INDEX=<Y|N>,
              IFLDNAME=field-name
              <,EXTINT=<INTERNAL|EXTERNAL>,
                EXTLEN=external-length,
                SPANNED=<Y|N>>),
     ASSOCED=(ASSOC=<Y|N>,
              AFLDNAME=field-name),
     SUBFILED=(SUBFILE=<Y|N>,
               SFLDNAME=field-name),
     SUBFIELD=(SUBFLD=<Y|N>,BASEFID=FIELDNAME,
               OFFSET=offset
               <,<FILE=<*filename|ANCHOR>>
               or <FILE=<ASSOCIATED|SUBFILE>,
                   FLDNAME2=field-name>>)
```

2. ADDLIKE
```
FLDNAME=new fieldname,
        FLDNAME2=other-fieldname
```

3. CHANGE
```
FLDNAME=field-name,
       TYPE=(FLDTYPE=field-type
             <,ALIGN=RIGHT|LEFT>>),
       FORM=(FLDFORM=field-format,
             FLDLEN=field-length,
             ELEMLEN=element-length,
             ELEMLIM=element-number
             <,UNIQUE=<Y|N>>),
       ROUTINES=(CONV=conversion-routine,
                 FORMAT=formatting-routine,
                 VALID=validation-routine,
                 VALIDARG=validation-argument),
       INDEXED=(INDEX=<Y|N>,
```

```
                IFLDNAME=field-name
                <,EXTINT=<INTERNAL|EXTERNAL>,
                  EXTLEN=external-length,
                  SPANNED=<Y|N>>),
        ASSOCED=(ASSOC=<Y|N>,
                AFLDNAME=field-name),
        SUBFILED=(SUBFILE=<Y|N>,
                 SFLDNAME=field-name),
        SUBFIELD=(BASEFLD=field-name,
        SUBFIELD=(SUBFLD=<7IN>,BASEFLD=FIELDNAME
                 OFFSET=offset
                 <,<FILE=<*filename|ANCHOR>>
                 or <FILE=<ASSOCIATED|SUBFILE>,
                     FLDNAME2=field-name>>)

4.  CHKPOINT   (none)

5.  CREATSUB FLDNAME=control-field-name,
             MAXRECS=#-records,
             ASSOC=<Y|N>,
             AFLNAME=field-name

6.  DELETE FLDNAME=field-name

7.  DISPLAY FLDNAME=field-name

8.  END   (none)

9.  FIELDS   (none)

10. FILE DESCOK=<Y|n>

11. FLDSEC FLDNAME=(field-name<,...>),
           SECURITY=(<<ADD|DELETE>.>
                     security-code<,...>)

12. MOVE FLDNAME1=new-location-field-name,
         FLDNAME2=field-name

13. PRINT   (none)

14. RECSEC DFLDNAME=field-name,
```

```
            SECURITY=(<<ADD|DELETE>.>
                      security-code:mask<,...>)
```

15. RENAME FLDNAME1=new-field-name,
FLDNAME2=old-fieldname

16. RESTORE (none)

17. SAVSTRT STRTNAME=strategy-name

18. SUPERFLD FLDNAME=field-name,
ROUTINES=(CONV=conversion-routine,
FORMAT=formatting-routine,
VALID=validation-routine,
VALIDARG=validation-argument),
FLDLIST=(<<INTERNAL|EXTERNAL>.>
field-name<,...>)

C. UPDATE MODE Command Formats.

1. CHANGE FLDNAME=field-name,
TYPE=(FLDTYPE=field-type
<,ALIGN=<RIGHT|LEFT>>),
FORM=(FLDFORM=field-format,
FLDLEN=field-length,
ELEMLEN=element-length,
ELEMLIM=element-number,
<,UNIQUE=<Y|n>>),
ROUTINES=(CONV=conversion-routine,
FORMAT=formatting-routine,
VALID=validation-routine,
VALIDARG=validation-argument)

2. DISPLAY FLDNAME=field-name

3. END (none)

4. FIELDS (none)

5. FLDSEC FLDNAME=field-name,
SECURITY=(<<ADD|DELETE>.>
security-code<,...>)

6. PATCH (field-name=value <,...>)

7. RECSEC [FLDNAME=field-name,
   SECURITY=(<<ADD|DELETE>,>
   security-code:mask<,...>

8. REVIEW FILE=file-name,
   FLDNAME=<*HEADER|FIELD-name>

APPENDIX B.

CREATE MODE

OPERAND RELATIONSHIPS

When creating descriptors there are certain implied relationships between the various operand combinations that may be specified. In those cases, the Descriptor Editor assumes the implied value and over-rides any value specified by the user. When modifying descriptors the Descriptor Editor normally interprets a default response to indicate no change to a particular operand.

The following table indicates the default values and the maximum values for several parameters of the ADD command.

TABLE 1

CREATE MODE

OPERAND DEFAULT AND MAXIMUM VALUES

FLDTYPE	FLDFMT	DEFAULT ALIGNMENT	MAXIMUM FLDLEN	MAXIMUM ELEMLEN	MAXIMUM ELEMLIM
A	F	L	3996-Key Length	NA	NA
A	V	L	3994-Key Length	NA	NA
A	FE	L	3994-Key Length	256	(FLDLEN)
A	VE	L	3994-Key Length	255	(FLDLEN/2)
B	F	L	1	NA	NA
BN	F	R	1	NA	NA
BN	FE	R	3994-Key Length	1	(FLDLEN)
BP	F	L	1	NA	NA
HX	F	L	2(3996-Key Length)	NA	NA
HX	V	L	2(3994-Key Length)	NA	NA
HX	FE	L	2(3994-Key Length)	256	(FLDLEN)
HX	VE	L	2(3994-Key Length)	255	(FLDLEN/2)
LN	F	R	4	NA	NA
LN	FE	R	3994-Key Length	4	(FLDLEN/4)
S	F	R	8	NA	NA
S	FE	R	3994-Key Length	8	(FLDLEN/8)
SD	F	R	5	NA	NA
SD	FE	R	3994-Key Length	5	(FLDLEN/5)
SN	F	R	2	NA	NA
SN	FE	R	3994-Key Length	2	(FLDLEN/2)
SS	F	R	4	NA	NA
SS	FE	R	3994-Key Length	4	(FLDLEN/4)

(1) Default conversion and formatting routine names are inserted by the editor unless specified by the user. The routine names have the format DBXXXYY, where;

"XXX" is either CVT for conversion routine or FMT for a formatting routine, and

"YY" is "SP" for field type "A" and is the field type itself for all other field types.

APPENDIX C.

PREDEFINED FIELDS

In most cases when the user defines or creates a new fieldname there is only one field descriptor created. There are, however, some exceptions to this which are enumerated below.

When the anchor file key field is completely defined by the user, the following fields are automatically defined and added to the list of field descriptors.

1. The FILEKEY field is a field defined over the anchor file key field. This field has all of the characteristics of the anchor file key field except for the field name and that it is a readonly field, that is a redefined field.

2. The fields FREEFORM and COMMENTS are defined for the retrieval system COMMENTS is a varying length field designed to hold any comment the user may wish to place there. FREEFORM will allow the user to specify his own particular keywords for the file he is referencing and he is able to base strategies on these user entered keywords.

The RECLEN is a predefine field which will appear in each descriptor region of the data base. This field defines the record length field which appears on each variable length record in a file.

When the user specifies record security for any file, for the first time, a field is created describing the record security code that appears in each data record of that file. This field is placed immediately after the anchor key for the anchor and associated files, and immediately after the parent key field on subfiles.

The record security fieldname is created in the following manner for the different file types:

1. ANCHOR file - the fieldname is RECSEC.

2. ASSOCIATED file - the fieldname is RECSEC concatenated to the suffix of the associated file, i.e. 1 to 9.

3. SUBFILE - the fieldname is the subfile control fieldname concatenated to RS.

When the user creates a subfile by the CREATSUB command the following fields are defined:

1. The subfile control field itself which resides either on the anchor file or an associated file.

2. The subfile key field which is the subfile control field name concatenated to ID.

3. The subfile parent key field which is a copy of the parent anchor key field. This fieldname is created by taking the subfile control fieldname concatenated with PK.

4. Allowance is made for subfile record security by creating the fieldname of subfile control field name concatenated to RS.

The field characteristics of each of the predefined fields are included in Table 2.

All of the aforementioned fieldnames are included in a reserved list. These fields cannot be altered by the user except in the following manner:

To modify FILEKEY, the anchor file key field must be modified. The predefined fieldnames for record security cannot be modified in any way and can only be created through use of the RECSEC command. The RECLEN field descriptor cannot be modified. The subfile control field and subfile key field cannot be modified once created. The subfile parent key field will only be changed to reflect changes in the anchor file key field. The fieldname for subfile record security can only be created through use of the RECSEC command.

Table 3 contains the names of the reserved fieldnames. As subfiles are created, the subfile control fieldname, the subfile key fieldname, the subfile parent key field name, and the subfile record security fieldname are placed in the reserved fieldname table, which then become reserved field names subject to the above listed restrictions.

TABLE 2

PREDEFINED FIELD CHARACTERISTICS

FLDNAME	COMMENTS	FILEKEY	FREEFORM	RECLEN	record[1] security	subfile[1] control	subfile[1] id	subfile[1] parent
ASSOCFIL	1	(none)	1	(none)	(none)	(5)	(none)	(none)
SUBFILE	(none)	(none)	(none)	(none)	(none)	(none)	(none)	(none)
INVFILE	(none)	(none)	A	(none)	(none)	(none)	(none)	(none)
READONLY	NO	Y	NO	YES	NO	YES	NO	YES
SUBCNTRL	NO	N	NO	NO	NO	YES	NO	NO
VARFLD	VARYING	F	VARYING	FIXED	FIXED	VARYING	FIXED	FIXED
BITFLD	NO	N	NO	NO	NO	NO	NO	NO
NUMALIGN	LEFT	(2)	LEFT	RIGHT	LEFT	RIGHT	RIGHT	(2)
VARELT	(none)	(none)	FIXED	(none)	(none)	FIXED	(none)	(none)
UNIQUELT	NO	(none)	NO	(none)	(none)	YES	(none)	(none)
INDEXEXT	(none)	(none)	INTERNAL	(none)	(none)	(none)	(none)	(none)
GENERCRT	DBCVTSB	(2)	DBCVTSB	DBCVTRL	DBCVTHX	DBCVTID	DBCVTID	(2)
VALIDRTN	(none)	(2)	(none)	(none)	(none)	(none)	(none)	(2)
REFORMAT	DBFMTSB	(2)	DBFMTSB	DBFMTRL	DBFMTHX	DBFMTID	DBFMTID	(2)
FLDPOSIT	2	4	1	0	(4)	(4)	4	7
FLDLEN	3988	(2)	3988	4	1	(6)	3	(2)
ELTLIM	0	0	100	0	0	(6)	0	0
ELTLEN	0	0	40	0	0	3	0	0
VALIDARG	(none)	(2)	(none)	(none)	(none)	(none)	(none)	(2)
NAMEFLD	(none)	(none)[3]	(none)[3]	(none)[3]	(none)[3]	(none)[3]	(none)[3]	(none)[3]
SECURITY	(none)[3]	(none)[3]	(none)[3]	(none)[3]	(none)[3]	(none)[3]	(none)[3]	(none)[3]

(1) Refer to the text for the derivation of the actual fieldname.
(2) The actual value is taken from the anchor key field.
(3) There is no field security on these fields unless specified by the user through use of the FLDSEC command.
(4) The value will be determined at "FILE" time.
(5) The value will depend on the "ASSOC" and "AFLDNAME" parameter values to the CREATSUB command.
(6) The actual value will depend on the input value to "MAXRECS" parameter to the CREATSUB command.

APPENDIX D.

DESCRIPTOR FILE OVERVIEW

Each descriptor file is an indexed sequential (ISAM) region Data Set where the key is developed by concatenating an eight character field name to a seven character file name. The name of the descriptor file is constructed by appending a "#" to the six-character data base name (padded with "$" if necessary).

The first record of each set of descriptors is called a header record and has a field name of blanks. This record is used by the system to reflect the current status and level of activity of that file, as well as controlling access to it, and is composed of fields described in Table 4. The remaining records are the field descriptors, themselves, and are composed of the fields described in Table 5.

TABLE 3

PREDEFINED RESERVED FIELDNAMES

1. COMMENTS
2. FILEKEY
3. FREEWORD
4. RECLEN
5. RECSEC
6. RECSEC1
7. RECSEC2
8. RECSEC3
9. RECSEC4
10. RECSEC5
11. RECSEC6
12. RECSEC7
13. RECSEC8
14. RECSEC9

TABLE 4

FILE DESCRIPTOR FIELD SPECIFICATION

FIELD NAME	FIELD TYPE	FIELD FORMAT	FIELD LOCATION	FIELD LENGTH	ELEMENT LENGTH	ELEMENT COUNT
RECLEN	LN	F	0	4	0	0
KEY	A	F	4	15	0	0
FLENAME	A	F	4	7	0	0
DATAPLEX	A	F	4	6	0	0
SUFFIX	A	F	10	1	0	0
FLDNAME	A	F	11	8	0	0
FILETYPE	A	F	19	1	0	0
DESCRCT	SN	F	20	2	0	0
BSELNGTH	SN	F	22	2	0	0
DESCOK	B	F	24	0(1)	0	0
SPANNED	B	F	24	2(1)	0	0
DATA	B	F	24	4(1)	0	0
MNTNABLE	B	F	24	6(1)	0	0
MNTNING	B	F	25	0(1)	0	0
LOADABLE	B	F	25	4(1)	0	0
REMAINS	HX	F	26	4	0	0
RECSECFP	SN	F	30	2	0	0
RSECTYCD	A	FE	1(2)	164	9	18

(1) For bit switches the length field is used to indicate the bit location within the byte.

(2) For variable length fields the location field is used as a variable field index.

TABLE 5

FIELD DESCRIPTOR FIELD SPECIFICATION

FIELD NAME	FIELD TYPE	FIELD FORMAT	FIELD LOCATION	FIELD LENGTH	ELEMENT LENGTH	ELEMENT COUNT
RECLEN	LN	F	0	4	0	0
KEY	A	F	4	15	0	0
FLENAME	A	F	4	7	0	0
DATAPLEX	A	F	4	6	0	0
SUFFIX	A	F	10	1	0	0
FLDNAME	A	F	11	8	0	0
ASSOCFIL	A	F	19	1	0	0
SUBFILE	A	F	20	1	0	0
INVFILE	A	F	21	1	0	0
READONLY	B	F	22	0(1)	0	0
SUBCNTRL	B	F	22	2(1)	0	0
VARFLD	B	F	22	4(1)	0	0
BITFLD	B	F	22	6(1)	0	0
NUMALIGN	B	F	23	0(1)	0	0
VARELT	B	F	23	2(1)	0	0
UNIQUELT	B	F	23	4(1)	0	0
INDEXEXT	B	F	23	6(1)	0	0
GENERCRT	A	F	24	8	0	0
VALIDRTN	A	F	32	8	0	0
REFORMAT	A	F	40	8	0	0
SPARE	HX	F	48	8	0	0
NAMECNT	SN	F	56	2	0	0
FLDPOSIT	SN	F	58	2	0	0
FLDLEN	SN	F	60	2	0	0
DFLDLEN	SN	F	62	2	0	0
ELTLIM	SN	F	64	2	0	0
DELTLIM	SN	F	66	2	0	0
ELTLEN	SN	F	68	2	0	0
DELTLEN	SN	F	70	2	0	0
VALIDARG	A	V	1(2)	52	0	0
NAMEFLD	A	FE	2(2)	146	8	18
SECURITY	A	FE	3(2)	]46	9	16

(1) For bit switches the length field is used to indicate the bit location within the byte.

(2) For variable length fields the location field is used as a variable field index.

APPENDIX E.

THE POSITION OF FIELDS WITHIN A RECORD

Fields are positioned in the data record in the order in which they are created as to the following algorithim. On the anchor and associated files the order is:

1. RECLEN,
2. anchor file key field,
3. record security field,
4. all packed bit fields,
5. all fixed length fields,
6. all varying length and elemental fields,

On subfiles the order by position is:

1. RECLEN
2. subfile key field,
3. subfile parent key field,
4. record security field,
5. all packed bit fields,
6. all fixed length fields,
7. all varying length and elemental fields.

The Descriptor Editor maintains three lists of fields for each descriptor region, one list for each of the folloiwng field groups:

1. packed bit fields,
2. fixed length fields including ordinary or unpacked bit fields,
3. varying length and elemental fields.

The order within each field group is determined by the order in which the user creates fields within each group. This ordering may be changed through use of the MOVE command.

TOPIC D.3 - DBLOAD USER's GUIDE

I. INTRODUCTION

The DBLOAD program is a generalized routine to be used for either initially loading data onto a newly defined file, or for updating an existing file. In either case, the descriptors for the file must have been completely specified by using the Descriptor Editor before any loading of data is attempted. The program is made general by the fact that each input record read is passed to a specifically written sub-routine which identifies each of the fields that comprise the record, and passes this information back to DBLOAD for processing.

This manual describes the mode of operation for DBLOAD, LINKEDIT for DBLOAD, and the parameters necessary to invoke it. The procedures to follow for writing a DBLOAD exit routine are also in this user's guide.

II. LINKEDIT

Since every load will have its own user-written exit routine, it is necessary to linkedit the new exit routine with other needed DBLOAD modules to produce the final executable DBLOAD module. A standard LINKEDIT deck could be made-up where all the user has to do is insert the INCLUDE card for his exit routine and then execute the LINKEDIT step.

As a separate step after the LINKEDIT, the DBTABLE procedure must be executed. This puts the new external entry point name of the DBLOAD Exit routine into a dictionary file that is used by NASIS.

Currently, the DBLOAD exit routine must be compiled with the PL/1 version F compiler to be compatible with DBLOAD.

Appendix B gives an example of a LINKEDIT for DBLOAD, with the DBTABLE step.

III. INPUTS AND OUTPUTS

DBLOAD Table 1 lists the major inputs and

outputs from the DBLOAD program.
and produced (output) by the DBLOAD
program.

INPUT

INPUT DATA SET: This data set contains the data that is to be loaded to the output databases. The input must be an indexed sequencial data set.

DATABASE DESCRIPTORS: This data set is the previously defined file descriptors for the database to be loaded.

PARAMETER CONTROL CARDS: This data set contains the program paramets for different program functions.

OUTPUT

DATABASE: This is the actual files making up the database. Database can include an anchor file, associated files, subfiles, and indexed files.

ERROR DATA SET: This data set is a copy of the input records that cannot be loaded to the database.

MESSAGE DATA SET: This data set contains error messages and other informational messages (such as record counts).

IV. CONTROL

The DBLOAD Program is controlled by job control statements. The job control statements are required to execute or invoke the DBLOAD program and to define the data sets that are used and produced by the program. The parameter statements are used to control the functions of the load.

JOB CONTROL STATEMENTS

DBLOAD Table 2 shows the job control statements necessary for executing or working the DBLOAD program.

DBLOAD Table 2. Job Control Statements for the DBLOAD program

STATEMENT	USAGE
JOB STATEMENT	This statement initiates the job.
EXEC STATEMENT	This statement specifies the program name (PGM = DBLOAD) or, if the job control statements reside in a procedure library, the procedure name.
SYSPRINT DD STATEMENT	This statement defines a sequencial message data set. The data set can be written onto a system output device, a magnetic tape volume or a direct access volume. (This DD statement must be present)
PRTOUT DD STATEMENT	This statement defines the informational data set written by the program. DCB = (RECFM=FA, LRECL=133, BLKSIZE=133). DDNAME fixed.
CARDIN DD STATEMENT	This statement defines the parameter card data set that contains the program parameters. Data set resides in the input stream. DDNAME fixed.
ERROR DD STATEMENT	This statement defines the error data set that will contain records that cannot be loaded to The database. DDNAME is not fixed.
DATABASE	These statements define all files that are

DD STATEMENTS	in the database (descriptors, anchor, associated, subfiles, indexes).
INPUT DD STATEMENT	This statement defines the input data set that contains data to be loaded to database. Must be an indexed sequencial data set. DDNAME not fixed.
STEPLIB DD STATEMENT	This statement defines the library where the production or latest version of the DBLOAD module resides.
STATIC DD STATEMENT	This statement defines the STATIC (statistics) file.
ESITAB DD STATEMENT	This statement defines the external static dictionary entry points file.

PARAMETER STATEMENTS

The DBLOAD program is controlled by parameter statements. The parameter statements are enterd in the input stream (CARDIN DD Statement) as required. Following is a list of DBLOAD program parameters with their functions:

'filename'
identifies the database to be loaded.

Specified as: a 1-6 character name of the database.

'mode'
identifies the mode of operation for the program

Specified as: a one character code, 'L' for load mode, 'U' for update mode, and 'R' for restart mode.

Default: load mode is assumed.

'exit'
identifies the name of the user exit routine which is to be called to describe the composition of each input record.

Specified as: a 1-7 character name of the user exit routine entry point.

Default: the exit name is constructed by prefixing the file name with an 'X'.

'format'
identifies the name of the key field reformatting routine.

Specified as: a 1-8 character name whose first character must be alphabetic and whose remaining character must be alphanumeric.

Default: no key formatting routine is assumed.

'anchor'
indicates whether the anchor file is to be loaded

Specified as: a one character code,

'Y' for yes, and 'N' for no.

Default: the anchor file will not be loaded.

'associate'
identifies the associate files to be loaded.

Specified as: a multiple element parenthesized list of associated file suffices (1,2,...9)

Default: no associated files will be loaded.

'subfile'
identifies the subfiles to be loaded.

Specified as: a multiple element parenthesized list of subfile suffixes (R,S,...)

Default: no subfiles will be loaded.

'index'
indentifies the fields to be inverted with this load.

Specified as: a multiple element parenthesized list of 1-8 character field names (FIELD1,FIELD2,...)

Default: no index files will be loaded.

'input'
identifies the fully qualified name of the input data set from which DBLOAD is to obtain its data.

Specified as: a 1-35 character fully qualified dataset name.
Default: no index files will be loaded.

'input'
identifies the fully qualified name of the input data set from which DBLOAD is to obtain its data.

Specified as: a 1-35 character fully qualified dataset name.

Default: the input dataset name is

constructed by appending the qualifier '.INPUT' to the file name.

'generate'
indicates whether or not large numeric keys are to be generated for the output data base.

Specified as: a one character code, 'Y' to indicate that large numeric keys are to be generated, and 'N' to indicate not to generate keys.

Default: Keys will not be generated.

'error'
identifies the fully qualified name of the error dataset to which invalid input records are to be dumped.

Specified as: a 1-35 character fully qualified dataset name.

Default: the error dataset name is constructed by appending the qualifier '.ERROR' to the file name.

'errors'
identifies the number of non-critical data errors that are allowed before terminating the program

Specified as: a 1-4 digit number.

Default: a limit of 100 errors is established.

Examples:

1. The user wants to load a file with the anchor, associated file 1, subfiles Y and Z. The key has a formatting routine entry point name of DBFMTLN. No fields are to be inverted. User exit routine is XEXIT and input file DSNAME is filename.input. Appendix A illustrates a run deck for above load.

V. OPERATING MODE

A. Load Mode

In the load mode, DBLOAD simply opens the input dataset for input and the file for

output and begins processing.

B. Update Mode

In the update mode, DBICAD opens the input file for input and the output file for direct output and begins processing.

C. Restart Mode

In restart mode, DBICAD opens the file for update. It uses the restart key to position itself in the input dataset. It then reads the next sequential record. It is now ready to begin processing.

VI. DBLOAD EXIT ROUTINES

A. Introduction

As mentioned earlier, DBICAD passes each input data record to a user written exit routine for analysis before actually writing any data to the file. This routine has the function of identifying each data field in the input record with a field name, indicating its starting location in the input record, and specifying the length of the data. If the data field is on a subfile, the exit routine has to identify the subfile control field name before any subfile fields can be put.
Further, the routine can specify that the field should have leading and/or trailing blanks stripped off by DBLOAD, that the field be skipped, that the record be skipped, that the load be terminated, or that subsequent calls to the exit routine must indicate when a new key is to be located to the output file. This is used in the case of multiple input records for an output file key.

When the update mode is used, the exit routine must indicate if this is a record to be deleted, a record to be added, or a record to be replaced.

B. Exit Routine Parameters

The calling sequence used by DBLOAD to transfer control to the exit routine is:

```
CALL exitname (input-data, user-ptr,
               bypass-switches)
```

Where:

'exitname'
     is the entry point name of the routine
     to be called.

'input-data'
     is a varying length character string
     (maximum size - 4000 bytes) that
     contains the entire input data record,
     including the four-byte record length.

'user ptr'
     is an external pointer that points
     to the user allocated structure.
     This structure contains the field
     names, the field lengths, the field
     offsets, and the subfile suffixes.

'bypass-switches'
     is a string of sixteen bit switches
     to be posted by the exit routine to
     further define the status of the record
     for DBLOAD. The order and meanings of
     the various bits are:
     Bypass Call - bypass subsequent calls
     Bypass Record - bypass this record
     Forward-Scan - delete leading blanks
     on fields
     Backward-Scan - delete trailing
     blanks on fields
     Terminate Pgm - terminate the program
     Delete-Record - delete this record
     Replace-Record - replace this record
     Update-Record - update this record
     (fields)
     New-Key - locate this new key
     Bits 10-16 - unused by DBLOAD

C. Exit Routine User Structure

The following sample exit routine (appendix C)

illustrates how to declare and use the
user based structure. First, set the
refer dimension equal to the maximum
number of fields and elements (one field
and a multi-element field with 10 elements
would be 11) plus number of subfile control

fields that may be assigned.

Next allocate the based user structure. Next assign the key-name, the key-ptr to the location of the key within the input record, and assign the key field length. Each entry into the exit routine will then require the field names to be assigned, the record, the field sizes assigned, and the subfile suffixes assigned if field is a subfile control field.

NOTES:

1. The Key of the input record can be anywhere in the record.

2. The input data record includes the record length field.

3. Large numeric keys can be generated for the output dataset if desired.
4. The number of elements in the user structure is computed by accumulating the total number of fields and/or elements in the input record.

5. Any field whose length is zero or whose pointer is null, is bypassed. If subfile suffix is not blank, new subfile record is located.

D. Sample DBLOAD EXIT Routine

The following sample exit routine is shown to illustrate the above narrative. The field has a key, one anchor file field, and two subfile fields with two elements each. The fields are all in fixed locations. After initial allocation, the only processing required is to scan a record type field for the code 'X' which is used to indicate to bypass this record. Note that all trailing blanks will be stripped off and that every input record is a new key and will have an output record located for it. The subfile control field 'KID' has a field size of zero. The sub-suffix byte for this field gets assigned a 'Z' to indicate this is a subfile control field.

VII. Loading Multi-files

For the most efficient use of DBLOAD, it should be noted that whenever possible, NC field should be inverted while file is being loaded. The DBPAC inversion process is extremely faster to load an entire file and then invert the desired fields with the inversion utility, invert. This module uses specialized techniques and an OS sort utility to build the index files.

Subfiles should never be loaded independently of the anchor file, however. DBPAC must generate the subfile keys and post the subfile control field for each subfile record.

APPENDIX A
SAMPLE JOB DECK

```
//CCCRDG        JOE (9350,SYST,060),NEOTERICS
//IBLOAD        EXEC PGM=DBLOAD,REGION=800K
//STEPLIB       DD DSN=NASIS.JCBLIB,DISP=SHR
//SYSPRINT      DD SYSOUT=A
//ERROR         DD DSN=FILENAME.ERROR,UNIT=2314
//               VOL=SER=WORK01,DISP=(NEW,KEEP),
//
DCB=(DSORG=IS,RECFM=V,IRECL=4001,BLKSIZE=4005,
//               RKP=5,KEYLEN=4,OPTCD=L),SPACE=(CYL,(3,1))
//INPUT         DD DSN=FILENAME.INPUT,UNIT=2314,
//               VOL=SER=WORK01,DISP=CID,DCB=(DSORG=IS,
//               RECFM=V,LRECL=4001,BLKSIZE=4005,RKP=5,
//               KEYLEN=4)
//FILE$$        DD DSN=NASIS.FILE$$FILE$$,DISP=SHR
//FILE$$#       DD DSN=NASIS FILE$$.FILE$$#,DISP=SHR
//FILE$$1       DD DSN=NASIS.FILE$$.FILE$$1,DISP=SHR
//FILE$$Y       DD DSN=NASIS.FILE$$.FILE$$Y,DISP=SHR
//FILE$$Z       DD DSN=NASIS.FILE$$.FILE$$Z,DISP=SHR
//STATIC        DD DSN=NASIS.STATIC,DISP=SHR
//ESDTAB        DD DSN1NASIS.LOADTAB,DISP=SHR
//CARDIN        DD *

               MODE=L
               EXIT=XEXIT
               FORMAT=DBFMTLN
               ANCHOR=Y
               ASSOCIAT=1
               SUBFILE=(Y,Z)
               INPUT=FILENAME.INPUT
               FILENAME=FILE$$

/* //
```

APPENDIX B
LINKEDIT FOR DBICAD

```
//CCCRDGLK      JOB(9350,SYST,015),NECTERICS,REGION=132K
//IKED          EXEC PGM=IEWL,
//              PARM='XREF,LIST,LET,SIZE=(150K,40K)'
//SYSPRINT      DD SYSOUT=A
//SYSLIB        DD DSN=NASIS.JCBLIB,DISP=SHR
//               DD DSN=ANSIS.TESTLIB,DISP=SHR
//               DD DSN=SYS1.PL1LIB,DISP=SHR
//NASIS         DD DSN=ANSIS.OBJLIB,DISP=SHR
//SYSLMOD       DD DSN=NASIS.TESTLIB(DBLOAD),DISP=SHR
//SYSUT1        DD DSN=&&SYSCT1,UNIT=SYSDA,SPACE=(1024,
//              (200)20)),SEP=(SYSLMOD,SYSLIB),DCB=BLKSIZE=
               1024 //SYSLIN        DD *
               INCLUDE NASIS (DBCALL,DBLOAD,PRTFILE,PPARM,
               DBMPAC)
               INCLUDE NASIS (DBUPDST,DBDBIO,JWEXITS,
               DBEXITX,DBRTNS)
               INCLUDE NASIS (EXITRTN)
               ENTRY IHENTRY

/* //DBTABLE       EXEC DBTABLE,MODULE=DBICAD,MODLIB=JOBLIB,
//              ESDTAB=LOADTAB
//RUNETAB.SORTOUT  DD DCB=NASIS.ESDTAB
//
```

APPENDIX C.

```
/*   XEXIT:  TEST EXIT ROUTINE FOR  RDBLOAD FOR THE FILE
DBWTDB       */

/*   COMPANY:  NEOTERICS CORPORATION, CLEVELAND, OHIO
*/
*/
/*   CLIENT:  NASA LEWIS RESEARCH CENTER
*/
/*   SYSTEM:  NASA AEROSPACE SAFETY INFORMATION SYSTEM
(NASIS)        */

/*        THIS IS A SAMPLE EXIT ROUTINE FOR DBLOAD FOR DB2.
IT       */
/*   SHOWS HOW TO USE THE USER STRUCTURE TO ASSIGN FIELD
NAMES, FIELD */
/*   OFFSETS, AND FIELD SIZES.  IT ALSO SHOWS HOW TO USE THE
EXIT     */
/*   ROUTINE SWITCHES TO ACCOMPLISH VARIOUS OPTIONS TO THE
LOAD.      */

XEXIT:    PROCEDURE(INPUT_DATA,USER_PTR,BYPASS_SWITCHES);

/*                  DECLARE BUILTIN FUNCTIONS USED BY EXIT
*/

     DCL (NULL,ADDR) BUILTIN:

*/

     DCL INPUT_DATA CHAR(4000) VAR;  /* INPUT RECORD
*/
     DCL USER_PTR POINTER;           /* USER POINTER
*/
     DCL BYPASS_SWITCHES CHAR (2);   /* PROGRAM SWITCHES
*/

/*                  DECLARE SPECIAL SWITCHES FOR PROGRAM
OPTIONS      */

     DCL 1 SPECIAL_SWITCHES BASED(SW_PTR),/* SPEC PROGRAM
SWITCHES    */
              2 BYPASS_CALL BIT(1),    /* SKIP FUTURE CALLS
TO EXIT  */
              2 BYPASS_RECORD BIT(1),  /SKIP THIS RECORD
*/
              2 FORWARD_SCAN BIT(1),   /* STRIP OFF LEADING
BLANKS   */
              2 BACKWARD_SCAN BIT(1),  /* STRIP OFF
TRAILING BLANKS  */
              2 TERMINATE_PGM BIT(1),  /ABORT THE LOAD
*/
```

```
                2 DELETE_RECORD BIT(1),  /* DELETE THIS
RECORD          */
                2 REPLACE_RECORD BIT(1)  /* REPLACE THIS
RECORD         */
                2 UPDATE_RECORD BIT (1), /* ADD THIS RECORD
*/
                2 NEW_KEY BIT (1),       /* LOCATE THIS NEW
KEY        */
                2 SPECIAL_FILL BIT(7);   /* UNDEFINED
*/

     DCL ON BIT(1) STATIC INIT('1'B);    /* ON BIT SWITCH
*/
     DCL OFF BIT(1) STATIC INIT('0'B);   /* OFF BIT SWITCH
*/
     DCL ONE_TIME CHAR(1) CONTROLLED;    /* ONE TIME SWITCH
*/

/*                    DECLARE USER STRUCTURE TO BE ALLOCATED
BY EXIT    */
/* ROUTINE AND DO THE FOLLOWING:
*/
/*    AND CONTROL FIELD AND ELEMENTS TO BE PUT     */
*/
/*         2.   ASSIGN FIELD NAMES.
*/
/*         3.   ASSIGN FIELD POINTERS TO OFFSET IN INPUT
RECORD         */
/*         4.   ASSIGN FIELD SIZES.
*/
     DCL 1 USER_STRUC BASED(USER_PTR),   /* USER BASED
STRUCTURE       */
                2 DIM FIXED BIN(15),     /* DIMENSION OF
ARRAYS         */
                2 KEY_NAME CHAR(8)       /* KEY NAME
*/
                2 KEY_PTR PTR,           /* KEY POINTER
*/
                2 KEY_SIZE FIXED BIN(15),/* KEY SIZE
*/
                2 ITEMS(DIM_REFER REFER  /* REFER DIMENSION
FOF ARRAYS */
                  (USER STRUC.DIM))      /*
*/
                3 FIELD_PTR PTR,         /* FIELD POINTER
*/
                3 FIELD_SIZE FIXED BIN(15), /* FIELD SIZE
*/
                3 SUB_SUFFIX CHAR(1);    /*SUFFILE
SUFFIX:FIRST SUB-  */
                                         /* FILE SUFFIX IS
CTRL FLD    */
     DCL DIM_REFER FIXED BIN(15);        /* REFER DIMENSION
```

```
TO BE ASSN */

/*                    DECLARE EXIT ROUTINE POINTERS
*/
     DCL PTR1 POINTER;                    /* POINTER FOR BASED
RECORD   */
     DCL SW_PTR POINTER;                  /* POINTER FOR
SWITCHES       */

/*                    DECLARE RECORD OVERLAY
*/

     DCL 1 RECORD_IN BASED(PTR1),         /* RECORD OVERLAY
*/
           2 RLTH     CHAR(4),            /* RECORD LENGTH
           2 RFILL    CHAR(3),            * FILLER
*/
           2 EMPNO    CHAR(4),            /* EMPLOYEE NUMBER
*/
           2 RTYPE    CHAR(1),            /* RECORD TYPE:IF
'X' THEN     */
                                          /* SKIP THIS RECORD
*/
           2 EMPNAME CHAR(20),            /* EMPLOYEE
NAME_STRIP OFF     */
                                          /*   TRAILING BLANKS
*/
           2 EMPAGE  CHAR(2),             /* EMPLOYEE AGE
*/
           2 KIDNAME1  CHAR(10),          /* KIDNAME_STRIP OFF
*/
           2 KIDNAME2  CHAR(10)           /*   TRAILING BLANKS
*/
           2 KIDAGE1   CHAR(2)            /* KIDAGE_STRIP OFF
*/
           2 KIDAGE2   CHAR(2);           /*   TRAILING BLANKS
*/
PTR1=ADDR(INPUT_DATA):                 /* SET RECORD POINTER
*/
SW_PTR=ADDR(BYPASS_SWITCHES);          /* SET SWITCHES POINTER
*/
IF ALLOCATION(ONE_TIME) THEN           /* IF USER STRUC
ALLOCATED          */
     GOTO CHECK_RECORD;                /*   GO AND CHECK RECORD
*/
ALLOCATE ONE_TIME;                     /* TURN ON ONE TIME
SWITCH          */
DIM_REFER_8;                           /* SET DIMENSION TO
MAXIMUM         */
                                       /* NUMBER OF FIELDS,
CONTRL         */
ALLOCATE USER_STRUC;                   /*   FIELDS, AND ELEMENTS
*/
```

```
DIM  =DIM_REFER;                        /* ASSIGN TO DIM IN STRUC
*/
KEY_NAM='EMPNO';                        /* ASSIGN KEY NAME
*/
KEY_PTR=ADDR(EMPNO)/* SET KEY POINTER  */
KEY_SIZE=4;                             /* ASSIGN KEY SIZE
*/

FIELD_NAME(1)='EMPNAME';                /* ASSIGN FIELD NAMES
*/
FIELD_NAME(2)='EMPAGE';                 /*
*/
FIELD_NAME(3)='KID';                    /* SUB RECORD ONE
*/
FIELD_NAME(4)='KIDNAME';
FIELD_NAME(5)='KIDAGE';
FIELD_NAME(6)='KID';                    /* SUB RECORD TWO
*/
FIELD_NAME(7)='KIDNAME';
FIELD_NAME(8)='KIDAGE';

FIELD_PTR(1)=ADDR(EMPNAME);             /* ASSIGN FIELD OFFSETS
*/
FIELD_PTR(2)=ADDR(EMPAGE);
FIELD_PTR(3)=NULL;                      /*     NULL CONTROL FIELD
PTR       */
FIELD_PTR(4)=ADDR(KIDNAME1);
FIELD_PTR(5)=ADDR(KIDAGE1);
FIELD_PTR(6)=NULL;                      /*     NULL CONTROL FIELD
PTR       */
FIELD_PTR(7)=ADDR(KIDNAME2);
FIELD_PTR(8)=ADDR(KIDAGE2);

FIELD_SIZE(1)=20;                           /* ASSIGN FIELD
SIZES          */
FIELD_SIZE(2)=2;
FIELD_SIZE(3)=0;
FIELD_SIZE(4)=10;
FIELD_SIZE(5)=2;
FIELD_SIZE(6)=0;
FIELD_SIZE(7)=10;
FIELD_SIZE(8)=2;

SUB_SUFFIX(1)=' ';                          /* ASSIGN SUBFILE
SUFFIX       */
SUB_SUFFIX(2)=' ';
SUB_SUFFIX(3)='Z';                          /*    SUBFILE Z
RECORD          */
SUB_SUFFIX(4)=' ';
SUB_SUFFIX(5)=' ';
SUB_SUFFIX(6)='Z';                          /*    SUBFILE Z
RECORD          */
SUB_SUFFIX(7)=' ';
```

```
SUB_SUFFIX(8)=' ';

 BYPASS_CALL=OFF;                    /* WILL CALL EXIT ROUTINE
*/
FORWARD_SCAN=OFF;                    /* DON'T STRIP OFF
LEADING BLANKS  */
BACKWARD_SCAN=ON;                    /* STRIP OFF TRAILING
BLANKS       */
TERMINATE_PGM=OFF;                   /* DON'T TERMINATE THE
LOAD         */
DELETE_RECORD=OFF;                   /* NOT AN UPDATE RUN
*/
REPLACE_RECORD =OFF;
UPDATE_RECORD=OFF;
NEW_KEY=ON;                          /* LOCATE THIS RECORD
*/
DISPLAY ('USER STRUCTURE ALLOCATED IN EXIT ROUTINE.');
RETURN;                              /* RETURN TO DBLOAD
*/

CHECK_RECORD:                        /* CHECK RECORD TYPE
*/
    IF RTYPE='X'                     /* IF X RECORD TYPE,
*/
        THEN BYPASS_RECORD=ON;       /*   BYPASS THIS RECORD
*/
        ELSE BYPASS_RECORD=OFF;      /* OTHERWISE, PROCESS IT
*/
     RETURN;                         /* RETURN TO DBLOAD
*/
END;                                 /* END OF ROUTINE
*/
```

TOPIC D.4 - INVERSION PROGRAM USER'S GUIDE

I. INTRODUCTION

The NASIS inversion program is two maintenance programs (DBIVRT1, DBIVRT2) and an OS sort utility for data base file creation. The purpose of the programs is to take data from certain fields of a database and to post this data to an inverted index file. This operation can be done automatically by DBPAC during a normal file loading operation, but it is very time consuming and could therefore jeopardize the successful completion of the load. Further, by separating this function out, in this manner, the capability of creating inverted indices after a file has been loaded and used is added to the repertoire of the NASIS system. Finally, this separation also permits the use of specialized techniques suitable specifically to this function to reduce the amount of time required for the entire process of loading and index creation.

This manual describes the mode of operation, invoking DBSIVRT, gives examples of use, and gives additional program notes.

II. MODE OF OPERATION

The inversion module can create up to ten inverted index files concurrently. Further, these files can each contain data from up to five separate but related fields. The user can process a specific number of input records, a range of input records, or the entire file. Restart capability is provided at the field reading step, the sort step, the index file creation step, and the index file translation step.

Step one (invert 1) reads a dataplex, strips off the field being inverted, concatenates the field with the anchor key, and writes the concatenated string on a sequencial data set.

The second, or sort step, invokes the OS sort utility and outputs a sorted sequential file.

Step three (invert 2) reads the sorted variable data set and creates a CISAM file. If the field is not indexed with external format, this file becomes the database index file.

If the field is indexed with external format

step three reads the QISAM file created by step three, translates the keys with field formatting routine, and creates translated index file.

III. INPUTS AND OUTPUTS

INVERT Table 1 lists the major inputs and outputs from the INVERT1 program.

INVERT Table 2 lists the major inputs and outputs from the INVERT2 program.

IV. CONTROL

The INVERT programs are controlled by job control statements and parameter statements. The job control statements are required to execute or invoke the INVERT programs and to define the data sets that are used and produced by the programs. The parameter statements are used to control the functions of the inversion.

JOB CONTROL STATEMENTS

INVERT table 3 shows the job control statements necessary for executing or invoking the INVERT process.

PARAMETER STATEMENTS

The INVERT process is controlled by parameter statements. The parameter statements are entered in the input stream (CARDIN DD Statement) as required. Following is a list of INVERT program parameters with their functions:

'FILENAME'
identifies the database that the field being inverted is on. Specified as a 1-6 character name.

'field'
identifies the field(s) to be inverted.

Specified as: a 1-8 character name as entered in the file descriptors. Multiple fields must be entered as multiple element list. Fields being inverted to same index file must be kept together.

Example: (A1,A2,A3,B1,B2,C) First three fields go on same index file, fields B1, B2 go on same index file, field C goes on

index file by itself.

'mode'
identifies the program mode of operation.

Specified as: a one character code,
F - initial pass, step one
R - restart at step one
3 - restart at step three (Step after Sort)
T - restart at translation phase of step 3

Default: the initial pass ('F') is assumed.

'records'
identifies the number of database records to process.

Specified as: 1-6 numeric characters.

Default: 999,999 records (or entire dataplex).

'range'
identifies a range of file keys to process.

Specified as: a multiple element list of two file keys, first key being the one to start at, second key being the one to end at.

Example: (KEY05,KEY10) Keys 5-10 will be inverted.

Default: Entire file is assumed.

INVERT Table 1. Data sets used (input) and produced (outputs) by the INVERT1 program.

INPUT	DATABASE: These data sets contain the data to be inverted. The descriptors are also needed to define the data fields.
	PARAMETER CONTROL CARDS: This data set contains the program parameters for different program functions.
	RESTART FILE: This data set is needed if program is invoked in restart mode, to provide a restart Key.
OUTPUT	OUTPUT FILE: This data set is a CSAM file with the value of the field being inverted concatenated with the file Key. This file becomes the input to the OS sort step.
	MESSAGE DATA SET: This data set contains error messages and other informational messages (such as record counts).

INVERT Table 2. Data sets used (inputs) and produced (outputs) by the INVERT2 program.

INPUT INPUT FILE: This data set is the sorted output from the OS sort utility step.

DATABASE DESCRIPTORS: This data set describes the database.

PARAMETER CONTROL CARDS: This data set contains the program parameters for different program functions.

OUTPUT PLEX FILE: This data set is in the form of an index file with the internal field value as the Key. This file becomes the input to the translation routine to convert to the external form. Produced only if external indexing.

RANGE FILE: This data set is the index file with the internal format. It is produced only if a range of file Keys was specified as a program parameter.

DATABASE INDEX FILE: This data set is the final index file and is part of the database.

MESSAGE DATA SET: This data set contains error messages and other informational messages (such as record counts).

INVERT Table 3. Jcb Ccntrol Statements for the INVERT process.

STATEMENT	USAGE
JOB STATEMENT	This statement initiates the job.
EXEC STATEMENT	This statement specifies the program name (PGM=INVERT1), (PGM=INVERT2), or, for the sort utility, the sort prccedure name (SORTD).
STEPLIB DD STATEMENT	This statement defines the library where the production or current version of the INVERT modules reside.
SYSPRINT DD STATEMENT	This statement defines a sequencial message data set. The data set can be written onto a system output device, a magnetic tape volume, or a direct access volume. (This DD statement must be present).
PRTOCT DD STATEMENT	This statement defines the informational data set written by the program. DCB=(RECFM=FA, LRECL=133, BLKSIZE=133). DDNAME is fixed.
CARDIN DD STATEMENT	This statement defines the parameter card set that contains the program parameters. Data set resides in the input stream. DDNAME is fixed.
DATABASE DD STATEMENTS	These statements define all the files that are in the database (descriptors, anchor, associated, subfiles, indices).
FILE DD STATEMENT	These statements define the QSAM files that are output from step one. Numeric integer 1-0 is concatenated to 'FILE' to make the appropriate DDNAME. Minimum LRECL is 18 (DSORT utility). LRECL is sum of file Key length plus maximum field length.
PARM DD STATEMENT	This statement defines the restart file for step 1. DCB = (RECFM=V, LRECL=255). DDNAME is fixed.
SORTIN DD STATEMENTS	These statements define the sorted QSAM data sets from the sort step. Numeric integer 1-Q is concatenated to 'SORTIN' to make the appropriate DDNAME.
PLEX DD	These statements define the temporary index files with the internal field

STATEMENTS format. The DCB will be the same as the final index file.

V. EXAMPLES OF USE

The INVERT process may be set up as one job with three separate steps (INVERT1, SORTD, INVERT2). If many fields are being inverted at one pass, however, it is recommended to split the three steps into separate jobs since these will be multiple sorts.

The most efficient method is to invert as many fields as possible in the same pass. Database records only have to be accessed one time for multiple fields. The multiple sorts could then be set up as separate jobs.

Invert all associated fields separate as one pass from anchor fields. This is very efficient because only the asscciated file is accessed.

Example one shows two fields, 'EMPAGE' and 'EMPNAME' being inverted at the same time. The database, 'FILE$$' has a Key length of 4. Since the EMPAGE field is only two bytes in length the minimum, LRECL of 18 is used for the FILE1 DD statement (2+4). The EMPNAME field is varying with a maximum length of 16, therefore, LRECL of 20 is used for FILE2 DD statement (4 + 16).

Step 2 is an example of the IBM OS sort utility, SORTD. For further explanation, see IBM publication order no. GC28-6543-7, CS SORT/MERGE Program. The sort control card specifies the sort field to start in first position of record and go for 6 bytes. The field is character and sort will be in ascending order. The size is 200 records. Note that if condition code from STEP1 is not less than 4, STEP2 will not be run. Only sort for the EMPAGE field is shown, sort for EMPNAME field would be similar.

Step 3 builds the final index files for the EMPAGE and EMPNAME fields. Note that if the sort step passes a condition code greater than 3 step 3 will not be run. A plex DD statement is needed for the EMPAGE field because of external indexing. The two INDEX DD statements catalog the index file entries after successful completion of the run. Note the MODE parameter is '3' for step 3.

Note that all data sets are deleted upon successful completion of job step. Only final index files for database should be kept.

IF range of Keys had been specified, final output index file would have a DSNAME of RANGE. FILE$$.FIELDNAME. This data set is then used to merge with other range index files.

```
//EXAMPLE1  JOB (9350,SYST,015), NECTERICS
//STEP1     EXEC PGM=INVERT1, REGION=500K
//STEPLIB   DD DSN=NASIS.JOBLIB,DISP=SHR
//SYSPRINT  DD SYSOUT=A
//PRTOUT    DD SYSOUT=A, DCB=(RECFM=FA,LRECL=133,
          BLKSIZE(=133) //DD#       DD
DSN=NASIS.FILE$$.FILE$$#, DISP=SHR
//DD        DD DSN=NASIS.FILE$$.FILE$$, DISP=SHR
//DDA       DD DSN=NASIS.FILE$$.FILE$$A,DISP=SHR
//DDB       DD DSN=NASIS.FILE$$.FILE$$B,DISP=SHR
//FILE1     DD DSN=SORTIN.FILE$$.EMPAGE,UNIT=2314
//         VOL=SER=WORK01,DCB=(RECFM=FB,LRECL=18,
//          BLKSIZE=3600), SPACE=(CYL,3),DISP=(NEW,
          KEEP,DELETE) //FILE2     DD
DSN=SORTIN.FILE$$.EMPNAME,UNIT=2314,
//         VOL=SER=WORK02,DCB=(RECFM=FB,LRECL=20,
//          BLKSIZE=4000),SPACE=(CYL,3),DISP=(NEW,
          KEEP,DELETE) //CARDIN    DD*
          FILENAME=FILE
          FIELD=(EMPAGE, EMPNAME)
          MODE=F /*
/
//STEP2     EXEC SORTD,REGION=98K,COND=(4,LT)
//SORT.SORTIN DD DSNAME=SORTIN.FILE$$.EMPAGE,
//         UNIT 2314,VOL=SER=WORK01,DCB=(RECFM=FB,
//          LRECL=18, BLKSIZE=3600), DISP=(CLD,DELETE,
          KEEP) //SORT.SORTOUT DD
DSNAME=SORTOUT.FILE$$.EMPAGE,
//         UNIT=2314,VOL=SER=WORK02,DISP=(NEW,KEEP,DELETE),
//
SPACE=(CYL,3),DCB=(RECFM=FB,LRECL=18,BLKSIZE=3600)
//SORT.SORTWK01 DD UNIT=2314,SPACE=(TRK,(10),,CONTIG)
//SORT.SORTWK02 DD UNIT=2314,SPACE=(TRK,(10),,CONTIG)
//SORT.SORTWK03 DD UNIT=2314,SPACE=(TRK,(10),,CONTIG)
//SORT.SORTWK04 DD UNIT=2314,SPACE=(TRK,(10),,CONTIG)
//SORT.SORTWK05 DD UNIT=2314,SPACE=(TRK,(10),,CONTIG)
//SORT.SORTWK06 DD UNIT(2314,SPACE=(TRV,(10),,CONTIG),
//          SEP=(SORTWK01,SORTWK02,SORTWK03,SORTWK4,
//          SORTWK05)
//SORT.SYSIN DD  *
          SORT FIELDS=(1,6,CH,A),SIZE=200
/*
//STEP3     EXEC PGM=INVERT2,REGION=500K,COND=(4,LT)
//SYSPRINT  DD SYSOUT=A
//STEPLIB   DD DSN=NASIS.JOBLIB,DISP=SHR
//PRTOUT    DD SYSOUT=A,DCB=(RECFM=FA,LRECL=133,
//          BLKSIZE=133)
//DD#       DD DSN=NASIS.FILE$$.FILE$$#,DISP=SHR
```

```
//DD          DD DSN=NASIS.FILE$$.FILE$$,DISP=SHR
//SORTIN1     DD DSN=SORTOUT.FILE$$.EMPAGE,
//           DISP=(CLD,DELETE,KEEP),UNIT=2314,DCB=
//            (RECFM=FB, LRECL=18,BLKSIZE=3600),
             VOL=SER=WORK02 //SORTIN2    DD
DSN=SORTOUT.FILE$$.EMPNAME,DISP=
//            (OLD,DELETE,KEEP),UNIT=2314,DCB=(RECFM=FB,
//            LRECL=20,BLKSIZE=4000),VOL=SER=WORK02
//PLEX1       DD DSN=PLEX.FILE$$.EMPAGE,DISP=(NEW,
//           DELETE),UNIT=2314,DCB=(DSORG=IS,RECFM=V,
//            LRECL=4001,BLKSIZE=4005,OPTCD=L,RKP=5,
//            KEYLEN=2),SPACE=(CYL,(3,1)),VOL=SER=WORK01
//INDEX1      DD DSN=NASIS.FILE$$A,UNIT=2314,DISP=(NEW,
//           CATLG,DELETE),DCB=(DSORG=IS,RECFM=V,
//            LRECL=4001,BLKSIZE=4005,RKP=5,KEYLEN=2,
//            OPTCD=L),SPACE=(CYL,(3,1)),VOL=SER=FDA011
//INDEX2      DD DSN=NASIS.FILE$$.FILE$$B,UNIT=2314,
//           DISP=(NEW,CATLG,DELETE),DCB=(DSORG=IS,
//            RECFM=V,LRECL=4001,BLKSIZE=4005,RKP=5,
//            KEYLEN=16,OPTCD=L),SPACE=(CYL,(3,1),VOL=
//            SER=FDA011
//CARDIN      DD *
            FILENAME = FILE
            FIELD = (EMPAGE,EMPNAME)
            MODE = 3
```

TOPIC D.5 - INDEX MERGE PROGRAM USER'S GUIDE

I. INTRODUCTION

The merger program (DBINDM) is a special program for the merging of index files. The purpose of the program is to take two index files (created from a like data base) and merge them to a new index file or inplace to the accepted current index file. Further, the user is given the options to process duplicate list elements.

This manual describes the mode of operation for DBINDM, the parameters used to invoke DBINDM, gives examples of its use, and gives additional program notes.

II. INPUTS AND OUTPUTS

DBINDM Table 1 lists the major inputs and outputs from the OBINDM program.

DBINDM Table 1. Data sets used (input) and produced (output) by the DBINDM program.

INPUT CURRENT INDEX FILE: This data set contains the current database index lists.

UPDATE INDEX FILE: This data set contains the new or update postings to be merged with current index file. This data base is usually created by the INVERT program and has a DSNAME of 'RANGE.'FILENAME.FIELDNAME.

DATABASE DESCRIPTORS: This data set provides information about the field that is indexed.

OUTPUT UPDATED INDEX FILE: This data set is the updated inplace version of the index file.

NEW INDEX FILE: This data set is the new merged index file created from the current index file and the update index file. The DSNAME is 'INDMRG.'FILENAME.FIELDNAME.

MESSAGE DATA SET: This data set contains error messages and informational messages (such as record counts).

III. CONTROL

The DBINDM program is controlled by job control statements and parameter statements. The job

control statements are required to execute or invoke the DBINDM program and to define the data sets that are used and produced by the program. The parameter statements are used to control the functions of the index merge.

JOB CONTROL STATEMENTS

DBINDM Table 2 shows the job control statements necessary for executing or invoking the DBINDM program.

DBINDM Table 2. Job Control Statements for the DBINDM Program

STATEMENT	USAGE
JOB Statement	This statement initiates the job.
EXEC Statement	This statement specifies the program name (PGM=DBINDM) or, if the jot control statements reside in a procedure library, the procedure name.
SYSPRINT DD Statement	This statement defines a sequential message data set. The data set can be written onto a system output device, a magnetic tape volume, or a direct access volume. (This DD statement must be present.)
PRTOUT DD Statement	This statement defines the informational data set written by the program. DCB= (RECFM=FA,LRECL=133,BLKSIZE=133). DDNAME is fixed.
CARDIN DD Statement	This statement defines the parameter card data set that contains the program parameters. Data set resides in the input stream. DDNAME is fixed.
DATABASE DD Statements	These statements define the database descriptors, and the current database index file.
RANGE DD Statement	This statement defines the update index file to be merged with the current index file. DSNAME must be 'RANGE.'FILENAME.FIELDNAME.
INDMRG DD Statement	This statement defines the new merged index file. The DSNAME is 'INDMRG.'FILENAME.FIELDNAME.
STEPLIB DD Statement	This statement defines the library where the latest version of the IBINDM program resides.

IV. MODE OF OPERATIONS

IBINDM can create a new inverted index file, or it can merge inplace to the current inverted index file. This is done at the discretion of the user.

V. INVOKING DBINDM

PARAMETER STATEMENTS

The DBINDM program is controlled by parameter statements. The parameter statements are entered in the input stream (CARDIN DD Statement) as required. Following is a list of DBINDM program parameters with their functions:

'FILENAME'
identifies the database with the index file being merged.

Specified as, a 1-6 character name of the database.

'Mode'
identifies the program mode of operation.

Specified as: a one character code,
'F' - FIRSTPASS
'R' - RESTART

DEFAULT: NONE.

'Mode1'
identifies the target merge file.

Specified as: a one character code,
'1' - New File
'0' - Inplace

'Field'
identifies the master inverted index field.

Specified as: 1-8 character name as entered in the file descriptors.

'Mode2'
indicates if duplicate list elements will be processed or not.

Specified as: a one character code,
'1' - Process Duplicates
'0' - No Duplicates Processed

VI. EXAMPLES

Following is an example of the job control

statements required to invoke DBINDM. The user wants to merge index file APOLLOA with RANGE.APOLLO.KEYWORDS and create a new index file with duplicates being processed. Field value is 4 bytes long.

```
//EXAMPLE1   JOB    (9350,SYST,060),NECTERICS
//STEP1      EXEC   PGM=DBINDM,REGION=800K
//STEPLIB    DD     DSN=NASIS.JOBLIB,DISP=SHR
//SYSPRINT   DD     SYSOUT=A
//PRTOUT     DD     SYSOUT A,DCB=(RECFM=FA,LRECL=133,
//                 BLKSIZE=133)
//DD$#       DD     DSN=NASIS.APOLLO.APOLLO#,DISP=SHR
//DD$        DD     DSN=NASIS.APOLLO.APOLLOA,DISP=SHR
//RANGE      DD     DSN=RANGE.APOLLO.KEYWORDS,DISP=SHR
//INDMRG     DD     DSN=INDMRG.APOLLO.KEYWORDS,
//                  UNIT=2314,VOL=SER=WORK01,
//                  DISP=(NEW,KEEP,DELETE),DCB=(DSORG=IS,
//                  RECFM=V,LRECL=4001,BLKSIZE=4005,
//                  RKP=5,KEYLEN=4,OPTCD=L),
//                  SPACE=(CYL,(3,1))
//CARDIN     DD     *
           FILENAME=APOLLO
           MODE =F
           MODE1=1
           MODE2=1
           FIELD=KEYWORDS /*
//
```

VII. PROGRAM NOTES

A. If the user wishes to merge inplace, he first must make a copy of the current index file (for security reasons).

B. The input or update index file to be merged is named RANGE.FILENAME.FIELDNAME. Check this before processing is begun.

C. When merging to a new file, the new file being created is called INDMRG.FILENAME.FIELDNAME.

D. After the new file is created and checked, it should replace the current index file, and the current index file should be erased.

TOPIC D.8 - MAINTENANCE USER'S GUIDE

I. INTRODUCTION

The maintenance program (DBMNTN) is an independent module to be used for maintaining the NASIS system database. This maintenance will consist of additions to, deletions from, modifications of the data contained on a database. The data to be used to describe the desired changes will take the form of transactions and will be obtained from the transaction database (TRNSCT). The transactions can reference a particular record, field or element in describing the desired change.

Maintenance, as designed, will always be run non-conversationally. It must be run under the userid of the owner of the database being maintained. The program is restartable in that, each transaction processed successfully, is deleted from the transaction database.

This manual describes the operating procedures, the mode of operation, and the types of transactions supported.

II INPUTS AND OUTPUTS

DBMNTN Table 1 lists the major inputs and outputs from the DBMNTN program.

DBMNTN Table 1. Data sets used (inputs) and produced (output) by the DBMNTN program.

INPUT DATABASE: These data sets are the files that make up the database including the descriptors.

TRNSCT FILES: These two data sets are the anchor transaction file and the TRNSCT file descriptors.

OUTPUT DATABASE: These data sets are the files of the database that will be updated.

MESSAGE DATA SET: This data set contains error messages and other informational messages (such as record counts).

III. CONTROL

The DBMNTN program is controlled by job control statements. The job control statements are

required to execute or invoke the DBMNTN program and to define the data sets that are used and produced by the program. The PARM statement is used on the EXEC card to define the database being maintained.

JOB CONTROL STATEMENTS

DBMNTN Table 2 shows the job control statements necessary for executing or invoking the DBMNTN program.

DBMNTN Table 2. Job Control Statements for the DBMNTN Program

STATEMENT	USAGE
JOB STATEMENT	This statement initiates the job.
EXEC STATEMENT	This statement specifies the program name (PGM=DBMNTN) or, if the job control statements reside in a procedure library, the procedure. The PARM function must be used with appropriate database name. (PARM = 'FILE$$')
SYSPRINT DD STATEMENT	This statement defines a sequencial message data set. The data set can be written onto a system output device, a magnetic tape volume, or a direct access volume. (This DD statement must be present).
PRTOUT DD STATEMENT	This statement defines the informational data set written by the program. DCB=(RECFM=FA,IRECL=133, BLKSIZE=133) DDNAME is fixed.
DATABASE DD STATEMENTS	These statements define all the files that make up the database (descriptor, anchor, associated, subfile, index).
TRNSCT DD STATEMENTS	These statements define the TRNSCT (transaction) file and the TRNSCT file descriptors.
STEPLIB DD STATEMENT	This statement defines the library where the production or latest version of the DBMNTN module resides.
STATIC DD STATEMENT	This statement defines the STATIC (statistics) file.

IV. MAINTENANCE OPERATING PROCEDURES

In preparing to run maintenance on a database, the Database administer should perform a preliminary step. He may use the CORRECT command to peruse the transactions and to delete any which he deems to be unnecessary or invalid (See CORRECT command User's Guide).

Once the transactions are determined to be acceptable, he is ready to initiate maintenance. Restart is similar, but should require no transaction editing.

V. MAINTENANCE MODE OF OPERATION

The database is opened for update, the transactions are opened for update and processing begins. Each transaction is handled separately and if successfully processed, the transaction is deleted.

VI. EXAMPLES

Following is an example of the job control statements to initiate the DBMNTN program:

```
//EXAMPLE1  JOB (9350,SYST,060),NEOTERICS,REGION=800K
//STEP1     EXEC PGM=DBMNTN,PARM='FILE$$'
//STEPLIB   DD DSN=NASIS.JOBLIB,DISP=SHR
//SYSPRINT  DD SYSOUT=A
//PRTOUT    DD SYSOUT=A,DCB=(RECFM=FA,LRECL=133,BLKSIZE=133)
//DD$#      DD DSN=NASIS.FILE$$.FILE$$#,DISP=SHR
//DD$       DD DSN=NASIS.FILE$$.FILE$$,DISP=SHR
//TRNSCT#   DD DSN=NASIS.TRNSCT.TRNSCT#,DISP=SHR
//TRNSCT    DD DSN=NASIS.TRNSCT.TRNSCT,DISP=SHR
//STATIC    DD DSN=NASIS.STATIC,DISP=SHR
//
```

TOPIC E.1 - TSPL/I LANGUAGE EXTENSION

I. INTRODUCTION

The terminal support preprocessor for NASIS (TS) allows the PL/I programmer to include in his program, statements, in normal PL/I syntax, which refer to and use the various terminal support functions. To enable the use of the TS preprocessor in a particular program, it is only necessary to insert the following statement:

```
% INCLUDE LISRMAC(TS);
```

This statement must appear before any actual use of the preprocessor itself.

The preprocessor functions available are listed in the appendix along with the terminal control block (TC) containing the various switches and control fields that are used by terminal support. The functions provided perform a set of generalized operations on the terminal device. These operations can be altered and refined by the setting of appropriate switches in the TC block before invoking the particular TS function. This alteration is most useful for the PUT and PROMPT operations.

In addition to the functions listed, terminal support has defined two interrupt conditions, ATTN and END, to facilitate programmer control of the terminal device. The ATTN condition is raised each time the user depresses the attention key on his terminal. When this occurs, terminal support calls the last defined PL/I ON block for ATTN's via the signal mechanism. If the ON block returns, terminal support will prompt the user for a command with the following message:

```
-ATTN:
```

The user may respond to this message with any of the "immediate" commands:

```
SYNONYM
SYNONYMS
DEFAULT
DEFAULTS
PROFILE
EXPLAIN
PROMPT
STRATEGY
KA
```

```
KP
BACK
END
APOFF
GO
```

A default response is interpreted as a GO. If during the execution of one of these commands, the user depresses the attention key, that command will be terminated and the user will be reprompted.

The user may define an ON ATTN block in his program, but he must adhere to the following restrictions:

1. He may only issue output TS functions.

2. If he wants to suppress the system prompt, he must branch out of his ON block (by so doing, he cannot return to the point of interruption).

If the user wishes to disable attentions completely, he must set the 'DISABLED' bit in the system data table USERTAB. (This action should only be taken in the most critical situations).

In the above description, if the user had responded with an END command, terminal support would have raised the END condition via the signal mechanism. The purpose of this condition is to provide a standard method of terminating a program or application and yet allowing it to perform any "clean-up" actions that are necessary. As with ATTN, any output TS messages will be allowed.

The terminal support functions assume that the device has a screen, and that this screen is divided into an upper output area and a lower prompting area. The logical dimensions of the screen are defined by the physical dimensions or the default values for the symbols SCRNHGT and SCRNWTH. The current dimensions of the screen can be found in the TC block during the execution of the program.

II. STATEMENTS

A. ENABLE <ATTN | END | *ALL>;

This statement causes the default coding for the END and/or ATTN conditions to be generated into the program. The default code for ATTN is to simply return to the point of interruption. The default code for END is to branch to a routine

that will terminate the program via a RETURN.

This statement, if present, must appear only once in the program and before any ENTRY statements. This statement also implies an ENTRY statement.

B. ENTRY;

This statement must be executed before any other TS statements, during a particular invocation of the program. It establishes the default ON blocks generated by ENABLE and calls terminal support to initialize the TC block. Because of its function, an ENTRY statement should appear at each entry point of a program, or at a common point in the processing for all entry points.

An ENTRY need not follow an ENABLE, as the ENABLE statement includes and implies ENTRY.

C. ON PAGE CALL(entry);

This statement establishes the name of the routine which is to process paging of the screen for the current function. When a function has filled the screen with data and terminates with more data to be displayed, a PAGE command will result in a call to the entry point specified by the most recent ON PAGE statement.

The "entry" parameter must be, or will be, converted to a character string of eight or fewer characters in length.

D. PROMPT MSG(key) <USING(inserts)> <INTO(buffer)>;

This statement has two functions, the outputting of a message (where the INTO clause is omitted) and prompting for a command. The message specified will be located in the message library and displayed to the user. Any inserts specified will be placed in the proper positions within the text before it is displayed. If the message cannot be found, terminal support will automatically issue a diagnostic containing the message key. If a command prompt is indicated, the text will be preceded by a dash (-) and a string of (": :") will be appended to it before it is displayed. All inserts will be stripped of leading and trailing blanks. Unspecified inserts will be replaced by "***".

The "key" parameter must be, or will be converted

to a character string of eight or fewer characters in length. The "inserts" parameter must be a list of twenty or fewer character strings. The "buffer" parameter must be a character string into which the command entered is to be placed. It should be eight characters in length, or greater.

If the command entered by the user, after synonym search, will not fit in the string specified by the user, TC.PROMPT.TRUNCATION will be turned on by terminal support. Further, this, or any other type of error (syntax, etc.), will cause TC.PROMPT.ERROR to be turned on.

E. PROMPT MSG(key)<USING(inserts)> KEYWORD(id) INTO(buffer);

This statement is used to request parameters and/or data from the user or from the profile.

The "key" parameter must be, or will be converted to a character string of eight or fewer characters in length. The "inserts" parameter must be a list of twenty or fewer character strings. The "id" parameter must be, or will be converted to, a character string of eight or fewer characters in length. The "buffer" parameter must be a character string into which the data is to be placed. The maximum size data element returned by terminal support is 255 characters.

If the TC.PROMPT.BYPASS bit is turned on by the user prior to this statement, terminal support will examine the remaining parameters in its buffer and the profile for the data value, but will not prompt the user. Otherwise, if no value is found in the buffer or in the profile, the user will be prompted for the data value. If the "id" parameter is null and the data is specified in keyword format, terminal support will post the keyword into TC.PROMPT.KEYWORD for the user. If the program detects an invalid data value and wishes to reprompt the user for it, the TC.PROMPT.ERROR bit should be turned on prior to the PROMPT. If any errors are encountered by terminal support, the TC.PROMPT.ERROR bit will be turned on. If the data entered will not fit into the string specified, the TC.PROMPT.TRUNCATION bit will be turned on. If the value returned was obtained from the user's profile, the TC.PROMPT.DEFAULT bit will be turned on. Likewise, if the value returned was a quoted

string, the quotes will be removed and the TC.PROMPT.STRING bit will be turned on. If the value returned is an element of a parenthesized list, only the element will be returned, and the TC.PROMPT.MORE_DATA bit will be turned on. Subsequent prompts will result in succeeding elements being returned, until the end of the list is reached.

F. READ INTO(buffer);

This statement causes the current contents of the screen to be returned to the user.

The "buffer" parameter must be a character string into which the data is to be placed.

G. WRITE FROM (buffer);

This statement causes the screen to be written from the area specified, without any editing.

The "buffer" parameter must be a character string which contains the data to be written.

H. PUT <LINE | PAGE> FROM(buffer) <TAG(value)> <FORWARD|BACKWARD>;

This statement causes a new record to be placed into the screen buffer.

The "buffer" parameter must be a character string which contains the data to be written. The "value" parameter must be a character string which is to be used to identify this output record. If LINE is specified, the record is sequentially added to the screen buffer. If PAGE is specified, the screen buffer is reset and this record becomes the first record of the new screen. The FORWARD and BACKWARD options are used to control the direction of the sequential filling of the screen buffer, from the top down, or from the bottom up.

If the user's data exceeds the width of the screen, the second and subsequent lines begin at the position indicated by TC.OUTPUT.INDENT. If the user's data causes the screen to overflow, the amount of data written is indicated by TC.OUTPUT.WRITTEN. If the user wishes only complete records to be written to the screen, he should have TC.OUTPUT.PUT_PARTIAL turned off. If the user wishes the screen to be automatically

written when the buffer is filled, he should turn on the TC.OUTPUT.AUTO_WRITE bit. If the user wishes to have his lines split between words (for text processing) he should turn on the TC.OUTPUT.WORD_BREAK bit. If the user has displayed a segment of the current record on the previous page and he wants the remaining segment tagged and/or indented, he must turn on the TC.OUTPUT.CONTINUATION bit. If the last PUT caused the buffer to be filled, the TC.OUTPUT.OVERFLOW will be turned on.

I. FLUSH;

This statement is used to force the contents of the screen buffer to be written, even though it is not filled. If the user wants to indicate that more data remains to be displayed via the paging mechanism, he should turn on the TC.OUTPUT.MORE_DATA bit before his last output operation.

J. FINISH;

This statement causes the preprocessor to generate the necessary code to enable execution time communication with terminal support. It must be the last TS statement in the program.

TOPIC G.1 - USAGE STATISTICS

I. INTRODUCTION

Usage Statistics is, essentially, a separate sub-system of NASIS, whose function is to collect and retain statistics, concerning the use and status of the system. The statistics maintained are divided into retrieval statistics, use of the system, and maintenance statistics, status of the data. The retrieval statistics include counts of the number of times that various commands have been invoked, the number of retrieval sessions, the dates and time used for those sessions, as well as the aggregate time spent retrieving data. The maintenance statistics include counts of the numbers of record additions, deletions and updates, for the anchor file, subrecord files and for all inverted index files.

The maintenance of these statistics is an automatic function and will not be discussed here. What will be covered by this document is the production and use of the reports available through Usage Statistics. It should be noted that the retrieval statistics are available to any NASIS user, while the maintenance statistics are available to the owner of the dataplex only.

II. STATISTICS CHECKPOINT

The statistics gathered for retrieval are maintained on a per session basis, with a capacity for thirteen sessions before re-initialization is necessary. Because of this, a check is made each time a new session is begun, and if re-initialization is necessary, a checkpoint listing of the retrieval statistics is produced, so that the data on file will not be lost.

The checkpoint report is a formatted list of the data on file for a particular NASISID, before reinitialization. It will contain a line entry for each of the sessions on record, displaying the command counts, the lines, the date, the file name, and other pertinent information. The DBA should examine this report to analyze the usage that NASISID is making of the system and of the individual dataplexes. If he deems that some action is necessary, e.g., a user is logged onto the system for excessive periods of time, but not executing many commands, he should do whatever he feels is required. In any event, the report should be retained for future reference and analysis, and

should probably be filed by NASISID.

A sample checkpoint report is included in Figure 1.

III. RETRIEVAL STATISTICS REPORT

By submitting JOB CCCRPRNTR, the status of the entire retrieval satistics file can be presented. This report displays the activity of the various NASISIDS, the various data bases and the various retrieval commands.

The retrieval report is formatted by NASISID, with a line entry for each terminal session. These entries present the various command counts, the lines, the file names, and other pertinent information. In addition, a summary is made, at the end, of the aggregate times and sessions for all users.

A sample retrieval report is included in Figure 2.

IV. MAINTENANCE STATISTICS REPORT

By submitting JOB CCCRPRTM, the status of the entire maintenance statistics file can be presented. This report should be used by the DBA to validate the maintenance records of each data base. In addition, it should be used to assess the maintenance activity of the various dataplexes. With this information, the DBA will be in a better position to know the exact status of his dataplexes, when to backup the system, when to reorganize his files, and many other questions that must be answered in order to maintain proper control over the system and its data.

The maintenance report is formatted by dataplex name, with a line entry for each maintenance run. These entires present the counts of the number of additions, deletions and updates made to the anchor and associated files, the subrecord files and the inverted index files. In addition, a summary is made, for each file showing the aggregate and the average number of additions, deletions and updates to the dataplex.

A sample maintenance report is included in Figure 3.

MAINTENANCE STATISTICS FOR SYSTEMS MANAGER ** 01/11/73 PAGE 1

DATAPLEX NAME	TOTAL TRNS	ANCHOR RECORDS	NUMBER RUNS	TRANS RUN	MAINTENANCE DATES	FILEPLEX ADDS	FILEPLEX DELETES	FILEPLEX UPDATES	SUBPLEX ADDS	SUBPLEX DELETES	SUBPLEX UPDATES	XPLEX ADDS	XPLEX DELETES	XPLEX UPDATES
ASRD1$		3,132	1		12/19/72	3,132								

FILEPLEX	ADDS	DELETES	UPDATES	
TOTAL	3,132			FOR ALL RUNS
AVERAGE	3,132			PER RUN

177

RETRIEVAL STATISTICS 01/03/73

NASISID	CONN-TIME HR:MM:SC	CPU-TIME HR:MM:SC:MS	# SES	STRAT LENGTH	STORED #	OWNER ID	FILE NAME	FIELD NAME	ACTUAL EXP	TOTAL SEL	NUMBER OF SRCH	CORR
NE01	0:53:30	0:00:48:790	5	0	0							
						SAOWNER	ASRD1$A	AUTHOR	3	0	0	0
						SAOWNER	ASRD1$B	KEYWORDS	13	0	0	0
						SAOWNER	DB2TDBA	EMPAGE	1	0	0	0
						SAOWNER	DB2TDBB	TOTALCAR	1	0	0	0
						SAOWNER	DB2TDBC	KIDAGE	1	0	0	0
						SAOWNER	DB2TDBD	PET	1	0	0	0
						SAOWNER	DB2TDBE	SVCDATE	1	0	0	0

178

SNAPSHOT (CHECKPOINT)OF RETRIEVAL STATISTICS RECORDS BEFORE REINITIALIZATION

12/18/72

LISR ID	CONN-TIME HR:MIN:SC	CPU-TIME HR:MN:SC:MS	# SES	STRAT LENGTH	OWNER-ID	FIELD NAME	FILE NAME	SESSION DATE	# EXPANDS	# SELECTS	# SEARCHS	# CORRECTS
NEO1	:19:40	0:00:12:399	2		SAOWNER	KEYWORDS	ASRDI$B	721215				
								721215		1		
								721215				
								721215				
								721215				
								721215				
								721215				
								721215				
								721215				
								721215				
								721215				
								721215				
								721215		1		